US Army Corps
of Engineers
Construction Engineering
Research Laboratories

USACERL Technical Report 96/58
April 1996

# A Methodology for Linking Symbolic and Graphical Models for Collaborative Engineering

by
Jeffery S. Heckel
Kirk D. McGraw
Michael P. Case

The demand for flexible and efficient design of new facilities and the maintenance of existing facilities has led to a new breed of tools for use by architects and engineers. These tools provide a symbolic model of the facility on which to base decisions and recommendations using knowledge and rules that are determined by domain experts within a particular field. However, these tools do not provide an easily understood graphical representation of the design as do traditional Computer-Aided Drafting (CAD) systems.

The objective of this research was the design and implementation of a methodology to allow efficient and reliable bidirectional communication between a rule-based analysis system and a CAD system. This methodology enables a symbolic model in the analysis system to be represented and manipulated from within the CAD system.

This report discusses the design decisions that were made during system research and development. It describes the methodology implemented and gives an example of its use. An appendix contains a list of functions that were implemented and used within the system.

The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products. The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

***DESTROY THIS REPORT WHEN IT IS NO LONGER NEEDED***

***DO NOT RETURN IT TO THE ORIGINATOR***

## USER EVALUATION OF REPORT

REFERENCE: USACERL Technical Report 96/58, *A Methodology for Linking Symbolic and Graphical Models for Collaborative Engineering*

Please take a few minutes to answer the questions below, tear out this sheet, and return it to USACERL. As user of this report, your customer comments will provide USACERL with information essential for improving future reports.

1. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which report will be used.)

_____

_____

_____

2. How, specifically, is the report being used? (Information source, design data or procedure, management procedure, source of ideas, etc.)

_____

_____

3. Has the information in this report led to any quantitative savings as far as manhours/contract dollars saved, operating costs avoided, efficiencies achieved, etc.? If so, please elaborate.

_____

_____

4. What is your evaluation of this report in the following areas?

   a. Presentation: _____

   b. Completeness: _____

   c. Easy to Understand: _____

   d. Easy to Implement: _____

   e. Adequate Reference Material: _____

   f. Relates to Area of Interest: _____

   g. Did the report meet your expectations? _____

   h. Does the report raise unanswered questions? _____

i. General Comments. (Indicate what you think should be changed to make this report and future reports of this type more responsive to your needs, more usable, improve readability, etc.)

_____

_____

_____

_____

_____

_____

5. If you would like to be contacted by the personnel who prepared this report to raise specific questions or discuss the topic, please fill in the following information.

Name: _____

Telephone Number: _____

Organization Address: _____

_____

_____

6. Please mail the completed form to:

Department of the Army
CONSTRUCTION ENGINEERING RESEARCH LABORATORIES
ATTN: CECER-TR-I
P.O. Box 9005
Champaign, IL 61826-9005

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. AGENCY USE ONLY *(Leave Blank)* | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | April 1996 | Final |

**4. TITLE AND SUBTITLE**
A Methodology for Linking Symbolic and Graphical Models for Collaborative Engineering

**5. FUNDING NUMBERS**
4A162784
AT45
FF-XS5

**6. AUTHOR(S)**
Jeffery S. Heckel, Kirk D. McGraw, and Michael P. Case

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
U.S. Army Construction Engineering Research Laboratories (USACERL)
P.O. Box 9005
Champaign, IL 61826-9005

**8. PERFORMING ORGANIZATION REPORT NUMBER**
TR 96/58

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Headquarters, U.S. Army Corps of Engineers (HQUSACE)
ATTN: CEMP-ET
20 Massachusetts Avenue, NW.
Washington, DC 20314-1000

**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**
Copies are available from the National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161.

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**
Approved for public release; distribution is unlimited.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT (Maximum 200 words)**

The demand for flexible and efficient design of new facilities and the maintenance of existing facilities has led to a new breed of tools for use by architects and engineers. These tools provide a symbolic model of the facility on which to base decisions and recommendations using knowledge and rules that are determined by domain experts within a particular field. However, these tools do not provide an easily understood graphical representation of the design as do traditional Computer-Aided Drafting (CAD) systems.

The objective of this research was the design and implementation of a methodology to allow efficient and reliable bidirectional communication between a rule-based analysis system and a CAD system. This methodology enables a symbolic model in the analysis system to be represented and manipulated from within the CAD system.

This report discusses the design decisions that were made during system research and development. It describes the methodology implemented and gives an example of its use. An appendix contains a list of functions that were implemented and used within the system.

**14. SUBJECT TERMS**
Computer-Aided Design and Drafting (CADD)
modeling
rule-based analysis

**15. NUMBER OF PAGES**
80

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | SAR |

# Foreword

This study was conducted for the Directorate of Military Programs, Headquarters, U.S. Army Corps of Engineers (HQUSACE) under Project 4A162784AT45, "Energy and Energy Conservation"; Work Unit FF-XS5, "Expert System Analysis and Concurrent Engineering for Energy Systems Design." The technical monitor was Dan Gentil, CEMP-ET.

The work was performed by the Engineering Processes Division (PL-E) of the Planning and Management Laboratory (PL), U.S. Army Construction Engineering Research Laboratories (USACERL). The USACERL principal investigator was Jeffery S. Heckel. Dr. Michael P. Case is Chief, CECER-PL-E; L. Michael Golish is Operations Chief, CECER-PL; and Dr. David M. Joncich is Chief, CECER-PL. The USACERL technical editor was Linda L. Wheatley, Technical Resources Center.

Recognition is due Josh Yelon (PL-E) for his early development and testing of the CBrain communication application. Steve Donoho (PL-E) is recognized for his development and testing of communication and manipulation code for use with Microstation CAD. Finally, recognition goes to Jerry LaGrou and Rajaram Ganeshan (PL-E) for their assistance in testing and making suggestions for this research.

COL James T. Scott is Commander and Acting Director, and Dr. Michael J. O'Connor is Technical Director of USACERL.

# Contents

# 1 Introduction

## Background

The use of Computer-Aided Drafting (CAD) systems has become commonplace within the architecture/engineering community (AEC). CAD systems provide an easily understood graphical representation of the design. Unfortunately, CAD systems do not provide a computable representation of the design (e.g., the computer cannot differentiate between a wall and a pipe because both are represented by line segments). Further, graphical models lack the depth necessary for analysis and evaluation.

In the Army, tight Federal budgets and a shift from global to continental-based forces require flexible and efficient design of new facilities and maintenance of existing facilities. These goals, coupled with a loss of experienced facility designers, demand more rigorous analysis and enhanced design quality from architects and engineers.

To meet the demand for higher-quality, lower-cost designs, a new breed of tools is emerging. These tools provide a symbolic model of the facility on which to base decisions and recommendations using knowledge and rules that are determined by domain experts within a particular field. While symbolic models are computable, they often are not easily understood.

This report defines a methodology for linking a symbolic model in a rule-based analysis system, the Agent Collaborative Environment (ACE), with a graphical model in a commercial CAD system. This methodology unifies symbolic and graphical models, allowing the user to create and manipulate the symbolic model from within the CAD system and vice versa.

## Objective

The objective of this research was to design and implement a methodology to allow efficient and reliable bidirectional communication between ACE and CAD systems. With this communication interface methodology established, a system was to be developed to enable objects within ACE to be represented within the CAD system.

## Approach

This problem was solved in two phases. First, a communication link was developed between the ACE and CAD systems. This link established a set of application programming interface (API) calls used to send messages between the two systems. Once the communication API was established, programmers developed a methodology for creating and manipulating the symbolic model in ACE based on the graphical model in the CAD system and vice versa. This methodology, called CADTalk, includes a protocol for displaying objects in the CAD system, a way to link these objects to form assemblies of objects, and the ability to manipulate these objects from within the CAD system and have the changes reflected within ACE. This methodology can be used by agent developers to create agents that build symbolic models from within the graphical environment (CAD system) and then perform analysis of this symbolic model within ACE.

## Mode of Tech Transfer

The ACE system is being used by several District offices to aid in architectural design and energy analysis of facilities. CADTalk is being used by some of these District offices with the ACE system to develop agents that will interface with CAD systems. The ACE system technology is being transferred to Gold Hill, Inc., Cambridge, MA through a Cooperative Research and Development Agreement (CRaDA). CADTalk is included in this CRaDA.

# 2 Problem Overview

## The Problem Domain

Current CAD systems are widely used in AEC for the display and representation of geometric graphical models. These systems allow a designer to easily and efficiently place and manipulate elements such as text, lines, arcs, and circles within a drawing. These drawing elements are used to represent design objects such as walls, doors, and windows. In addition to the graphical information displayed by the CAD system, many objects also have some type of associated symbolic information. For example, a wall can be represented in the CAD system as a series of lines, but there is also symbolic information related to a wall, such as its construction type, R-value, and fire rating. Some CAD systems allow this type of information to be stored in an external database or as graphical element "attributes." However, these systems are unable to represent conceptual objects such as spaces and rooms. Further, they do not support relationships between objects. Without such relationships, it is very difficult to determine if two pipes are part of the same system or two doors are part of the same wall.

## The Symbolic Model

The symbolic model for an object consists of attributes and data that are not geographical or graphical in nature. Data of this type is better suited for reasoning analysis and evaluation. The ACE system is an agent-based software environment developed at USACERL. Agents are expert systems that are tightly integrated with each other using libraries of objects such as walls, fans, or pumps. The primary role of an agent in ACE is as an assistant that uses heuristic rules and a checklist facility to automate routine tasks. Experienced users can store their knowledge in agents for use by others. With this knowledge and a symbolic model created in ACE, a user can easily perform complex or repetitive analyses of the model. This analysis can then be used to alert the user of errors in the model and even alter the model to correct these errors.

## The Graphical Model

The graphical model is composed of data that can be used to visualize an object. These data allow architects/engineers to better visualize their design. Commercial CAD systems have been designed to store and manipulate this type of data efficiently, with interfaces that allow a user to create complex drawings and display them. However, the information contained within these systems is difficult to analyze other than graphically. For example, two line elements within a drawing might be identical except for their separate geographic or graphical information. In this case, a CAD system could not distinguish an interior wall from an exterior wall or a wall from a pipe.

## The Generic Symbolic/Graphical Architecture

The developed system consists of separate generic symbolic and graphical models. The primary reason for this separation was the desired ability to interact with several different systems. A generic symbolic model can be communicated to other programs, which can perform specific analyses of their own. Similarly, the generic graphical model can be moved or displayed in different CAD systems.

# 3 CBrain: The Communication Link

The first requirement in the integration of the ACE symbolic model and the CAD graphical model was establishing a communication link between the symbolic and graphical environments. This link would need to provide communication between ACE and each of the two CAD systems in widespread use by Corps District offices—AutoCAD® by Autodesk* and Microstation® by Bentley Systems. ** This communication link needed to be efficient, easy to use, and, most importantly, reliable. CBrain was the system developed to meet these requirements.

The low-level method of communication between ACE and the CAD system is done using the Microsoft® Windows® Dynamic Data Exchange (DDE) mechanism that sends data from one application to another. A client application makes a connection to a server application and a channel is established. With this channel established, the client application can then request information from the server application. These requests are strings of data that the server application then interprets and responds to.

A concern with the interaction between ACE and the CAD system was the possibility that a request from CAD to ACE would in turn require a request from ACE to CAD. This possibility could continue recursively several times, adding to the complexity of a single request. In this situation, it is important to match incoming responses with the appropriate outgoing request. If this matching is not maintained, the system could become corrupted.

To address this concern, the DDE messages sent between the two systems are not simple strings of information. A message from one system to the other contains three separate "packets" of information joined into one message. The three packets of information correspond to:

*operator* - The operator defines the type of message transmitted. The available types are an *evaluation*, a *result*, or a *cancel*.
*sequence number* - The sequence number is used to coordinate among recursive levels of requests.

---

* Autodesk, Inc., 111 McInnic Parkway, San Rafael, CA 94903.
** Bentley Systems, Inc., 690 Pennsylvania Drive, Exton, PA 19341-1136.

*arguments* - The arguments are used with the operator to perform the desired operation. The arguments most often would be the command to be evaluated.

Messages containing packets are sent from one system to the other requesting the value of an item or execution of a function. A message is decoded, processed, and the result sent back. An example of a message from CAD to ACE might be: "EVAL 1 (+ 1 2)." This message would be decoded and the arguments "(+ 1 2)" sent to the Lisp "EVAL" command. The result would then be sent back in the message: "RESULT 1 3." Notice that the sequence number is the same in both the request and result messages. Although messages are normally sent from CAD to ACE, it is possible for a message to be sent in the reverse direction, from ACE to CAD. Therefore, ACE and CAD act as both server and client applications at times during program execution.

Two low-level functions are defined for use with the CBrain communication interface on both the ACE and CAD sides. These functions are *ac-eval* and *ac-eval-text* on the ACE side and *cl-eval* and *cl-eval-text* on the CAD side. Functions *ac-eval* and *cl-eval* accept a command as a parameter, execute it on the other system, and return a result. For example, *(ac-eval '(+ 1 2))* would return the result *3*. Functions *ac-eval-text* and *cl-eval-text* accept a string as a parameter, pass it to the other system, read it, and return a result. For example, *(ac-eval-text "(+ 1 2)")* also would return the result *3*. All other functionality of the ACE/CAD linking is built upon the use of these basic functions to execute commands on each of the systems. CBrain works both in the Microsoft® Windows® and Microsoft® Windows NT™ operating systems.

# 4 Linking of the Symbolic and Graphical Models With CBrain

Once the communication interface was established between ACE and the CAD system, it was possible to link symbolic objects in ACE with graphical objects in CAD. To create this link, a unique identifier from ACE is associated with the object in CAD, and likewise, a unique identifier in the CAD system is associated with the object in ACE. Once this link is set up, operations on an object in one system will result in the notification of the associated object in the other system.

Figure 1 illustrates the creation of a link between the symbolic and graphical data. An instance of a *WALL* symbolic frame definition is created with the name *EAST-WALL* in ACE. This instance is represented in the CAD system as a set of line entities. Each graphical object within the CAD system (AutoCAD) has a unique identifier called an entity handle. User data (called extended entity data) may be attached to a graphical object within AutoCAD. After the series of lines is drawn

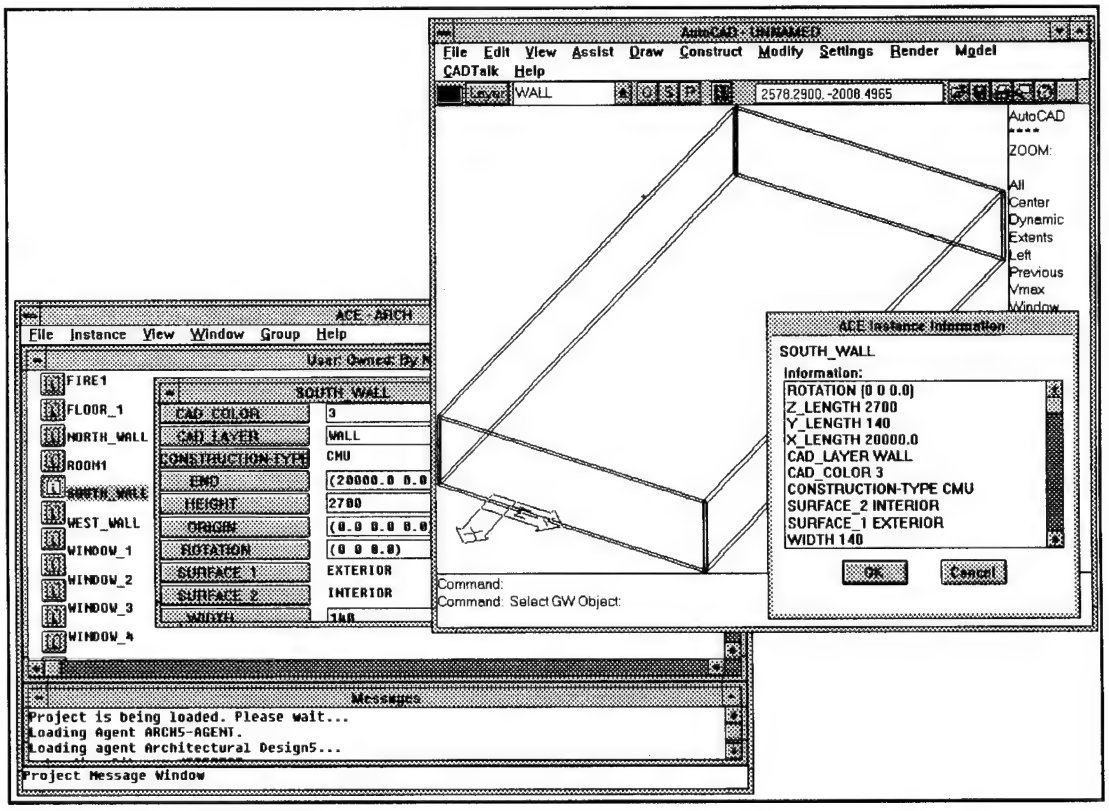


**Figure 1. Example linkage.**

within AutoCAD, the name of the object in ACE, *EAST-WALL*, is added to the graphical objects as extended entity data. In turn, the entity handles for the lines are added to a slot in the *EAST-WALL* object in ACE. With this process complete, a link is established between the ACE object and its associated CAD object. The same procedure is used in Microstation using a tag as the unique identifier and linkage information as the user data.

It is now possible to automate this process and provide extended capabilities to these objects. An example of an extended capability is the ability to manipulate objects within the CAD system and have these changes reflected in ACE. This was done by writing specialized forms of native commands for the CAD system, such as the *move* command. This specialized command would perform the movement of objects in the CAD system and then inform each associated instance in ACE using a message sent from CAD to ACE. This message contains the relative distance that the object was moved within the CAD system. The ACE system could either record this information, disallow the move, or perform some analysis based on the new object location.

# 5 Issues in the Use and Implementation of Linking

This chapter covers several items of concern that arose during the design of the linking methodology. These concerns had to be resolved before the actual coding of the implementation could begin.

## Compound Objects

An issue that was initially addressed was how to relate symbolic objects that need to be graphically linked. Several different objects may be linked to form a higher level object or an assembly. Certain actions performed on the higher level object may, in turn, require an operation to be performed on lower level objects. A wall, for example, may contain several doors and windows. When the wall is moved, these doors and windows must move as well. This linking of objects was solved using a semantic link within ACE called *HAS-PART*. A semantic link is a structure for creating an arbitrary relationship between two separate objects within ACE. *HAS-PART* is one type of semantic link within ACE. Other types can be defined by the user for different purposes, such as a *SUPPORTED-BY* semantic link for a floor and beam object. The *HAS-PART* semantic link has a *FROM* object and a *TO* object. In this example, the wall object *HAS-PART* a door and several windows. The link goes from the wall to the door and windows. Therefore, when an object is moved and has a *HAS-PART* semantic link to other objects, those objects are moved as well. Using the *HAS-PART* semantic link, it is possible to create a model of how a set of objects are linked together to create an assembly. A building, for example, can be defined with the model shown in Figure 2, established using several *HAS-PART* semantic links between different building component objects.

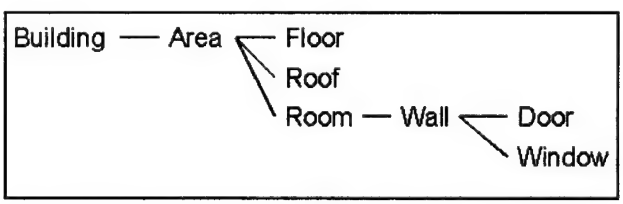


**Figure 2. Building has-part object model.**

## Local Coordinate System

Another issue that arose concerned the geometric relationship from one object related to another. Considering objects related using the *HAS-PART* semantic link, the lower object in the *HAS-PART* relationship should be geometrically placed in relation to the higher level object. To address this concern, all CAD-aware objects have an origin and rotation value associated with them. These two values are used to create a local coordinate system (LCS) for that object. Figure 3 shows the LCS for a *WALL* object. A *WALL* object, for example, is placed relative to the location of a specific *ROOM* object within the model presented in Figure 2. In turn, the *ROOM* object is placed relative to the *BUILDING* object. Only the top-most object in the *HAS-PART* relationship hierarchy (a *BUILDING* object in this case) has a true, absolute geometric origin stored with it.

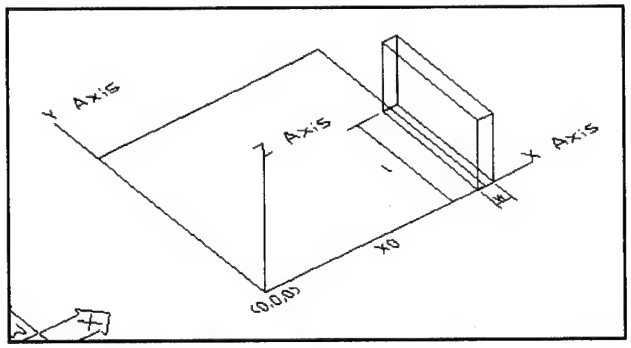

**Figure 3. Wall local coordinate system.**

With the LCS established for an object, it is possible to then build a local transformation matrix for that object. To derive the global coordinates for an object, it is necessary to build the transformation matrix for an object, move up the HAS-PART relationship to the higher object, build its transformation matrix, multiply the two matrices together, and continue up the HAS-PART relationship until no further objects are available. The final matrix will be the global transformation matrix for that object. Multiplying a local three-dimensional point through the global transformation matrix will result in a global three-dimensional point. The functions for these operations are defined in ACE Lisp file *cadshape.lsp* (see Appendix A for listing). Several functions have been written to hide this complexity from the user, they are:

```
;*****************************************************************************
;gets the transformation matrix of an instance - traces down through linked items
(define-handler (CAD-SHAPE-OBJECT get-matrix) ()

;*****************************************************************************
(define-handler (CAD-SHAPE-OBJECT get-relative-origin) ()

;*****************************************************************************
;get an objects absolute origin - if there is a reference object, then get it's transformation
;matrix and multiply our origin by that
(define-handler (CAD-SHAPE-OBJECT get-absolute-origin) (&aux temp)
```

```
;*****************************************************************************
(define-handler (CAD-SHAPE-OBJECT set-relative-origin) (new)

;*****************************************************************************
;set an objects absolute origin - if there is a reference object, then get it's transformation
;matrix and multiply the new origin by the inverse of that
(define-handler (CAD-SHAPE-OBJECT set-absolute-origin) (new &aux temp)

;*****************************************************************************
(define-handler (CAD-SHAPE-OBJECT get-relative-rotation) ()

;*****************************************************************************
(define-handler (CAD-SHAPE-OBJECT get-absolute-rotation) (&aux temp)

;*****************************************************************************
(define-handler (CAD-SHAPE-OBJECT set-relative-rotation) (new)

;*****************************************************************************
(define-handler (CAD-SHAPE-OBJECT set-absolute-rotation) (new &aux temp)
```

## Generic CAD Shapes

Once the methodology had been established for creating links between symbolic objects within ACE and graphical objects within CAD, basic geometric shape objects were created within ACE that would contain much of the necessary code for implementing CAD-aware objects within ACE. These shapes were simple could be represented in multiple CAD systems and manipulated within them. The first initial shapes were text, line, two-dimensional polygon, three-dimensional polygon, arc, and sphere. With this basic coding completed, it was possible for a user to create an object that inherits from these shape objects and receives all needed information to be graphically represented in CAD. For example, a *WALL* object definition can inherit from the three-dimensional polygon object to acquire its CAD graphical representation. These CAD shape objects have common attributes such as an origin, rotation, and possibly length in each of the x, y, and z directions of a Cartesian axis system. Figure 4 shows a three-dimensional polygon object.

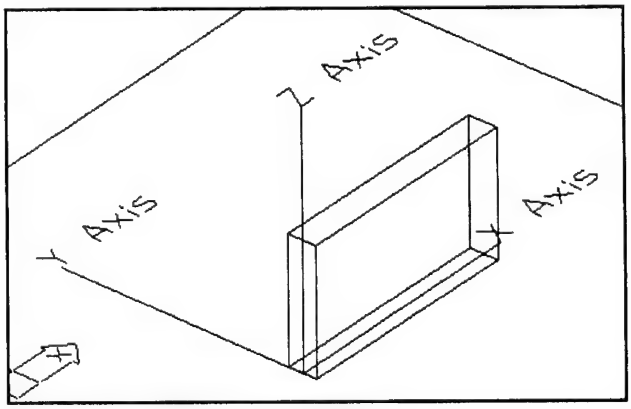


**Figure 4. 3D polygon representation.**

# 6 Methodology for Manipulating and Displaying Symbolic Objects Within a CAD System

With the communication interface between systems established, a procedure for creating a link between the symbolic object in ACE and the graphical object in CAD defined, and issues about linking and relating symbolic objects in ACE addressed, it was time to address how to actually manipulate and display an ACE symbolic object in the CAD system. To accomplish this, object methods (called handlers) were written in ACE in the Lisp programming language. All symbolic frames in ACE descend from the frame *BB-OBJECT*. By creating CAD-specific slots and handlers for this frame, all descendants inherit these slots and handlers. These handlers have generic names and implementations that then call specialized handlers based on the current CAD system used by ACE.

In many situations, a frame may not be representable within a CAD system. To prevent an attempt to draw a nondrawable object, *BB-OBJECT* was given a slot named *DRAWABLE*. By default, this slot value is *false*. If a user defines a new frame with a default value of *true* for this slot, the frame is considered CAD-aware. Two handlers were written for *BB-OBJECT* to "get" and "set" this value, *get-drawable* and *set-drawable* respectively.

The *BB-OBJECT VIEWS* slot contains the unique CAD identifier associated with the graphical representation within the CAD system. This slot contains an association list, which contains multiple occurrences of a pair containing a key and associated value. For example, ((key1 value1) (key2 value2)), is an association list with two pairs shown. This association list allows ACE to track the graphical representation of a symbolic object within multiple CAD systems at once. One key is associated with AutoCAD and another key is associated with Microstation. In the future, other CAD systems can be added as well. The generality of the *VIEWS* slot also can be used in a system other than CAD to represent a symbolic object, which could be better represented by a spreadsheet file. The *VIEWS* slot may contain an entry such as: (EXCEL "file1.xls"). It would then be possible to start Microsoft® Excel® and load spreadsheet "file1.xls" to display information relevant to this symbolic object. Three

handlers were written for *BB-OBJECT* to deal with this slot value: *set-view-tag*, *get-view-tag*, and *clear-view-tag*.

Several functions and handlers are defined for the actual creation and display of a symbolic object. Creation of a new CAD-aware symbolic object from the CAD system is performed with a call to the ACE function *process-drawable-object.* This function accepts the name of a frame and an agent as parameters. A new object is created using the given frame name definition and assigned to the given agent. A handler, *cad-create* (defined under *BB-OBJECT*), is then called and used to acquire initial values for a newly created object before being displayed. The handler returns *true*, indicating success.

Finally, the handler *cad-draw* is called to display the object. This handler (also defined under *BB-OBJECT*) would call a function or invoke a command in the CAD system to display the object. The drawing of the object typically uses previously set slot values for the object, hence the need for the cad-create handler. Once the object is displayed in CAD, symbolic object name must be associated in ACE with the unique identifier in CAD. The CAD system should return this unique identifier to ACE. Then two handlers, *set-cad-inst-name* and *get-cad-inst-name*, can be called to form the associations. Once this association is set, a symbolic object can be displayed in the CAD system.

The next requirement is an ability to manipulate the object in one system and have the changes reflected in the other. To accomplish this, two things need to be done, one in the ACE system and the other in the CAD system. In the ACE system, "when-modified" functions need to be assigned to slots with values relevant to the graphical representation. When-modified functions are associated with a slot and are automatically called when the slot value is modified. Typically, when a relevant slot value changes, the *cad-draw* handler will be called to redisplay the object.

In the CAD system, native commands must be rewritten or a user interface written to allow manipulation of these special objects. The move command, for example, is rewritten in AutoCAD so it can deal with these special objects. If one of these objects is moved, a message is sent to ACE informing it of the move, and the slot values are updated in the symbolic object.

# 7 Example Usage

This chapter gives an example of the set up and use of symbolic and graphical model linking using ACE and AutoCAD. First, the initial work necessary to make a symbolic ACE object CAD-aware will be described. Next, how to start the ACE/CAD interface using predefined ACE functions will be demonstrated. This chapter assumes a knowledge of the Lisp (Steele 1990) programming language, the ACE (Hoff et al. 1995) system, and the Goldworks® (Gold Hill 1990) frame system. The functions and handlers used within this example are described in Appendix A.

## Symbolic Frame Definition

To implement the ability for a symbolic object in ACE to be represented in a CAD system, the object must "know how" to display itself in CAD. Two approaches for this are: (1) inherit from one of the generic base CAD shapes that have been predefined or (2) create a custom drawing function in the CAD system that the symbolic object will call for display within CAD.

### *Inherit From A Base CAD Shape*

The easiest and fastest approach for making a CAD-aware symbolic frame is to inherit from one of the predefined base CAD shape objects. New frame definitions inherit from these base CAD shape objects and are then specialized with the appropriate slots. Base CAD shape frame definitions are located in the ACE Lisp file *cadshape.lsp* (see Appendix A for listing). This file must be added to an ACE library for use of the base CAD shape frame definitions. The frame definitions for *CAD-OBJECT*, *CAD-SHAPE-OBJECT*, and *3D-CUBE-CAD-SHAPE-OBJECT* are defined as:

```
(define-frame CAD-OBJECT
    (:is BB-OBJECT)
    (DRAWABLE :default-values (t))
    (ORIGIN :default-values ((0 0 0))
        :when-modified (cad-update-linked))
    (ROTATION :default-values ((0 0 0))
        :when-modified (cad-update-linked))
    (CAD  COLOR :default-values (1))
```

```
    (CAD_LAYER :default-values (0))
)

(define-frame CAD-SHAPE-OBJECT
    (:is CAD-OBJECT)
)

(define-frame 3D-CUBE-CAD-SHAPE-OBJECT
    (:is CAD-SHAPE-OBJECT)
    (X_LENGTH
        :doc-string "X length relative to origin."
        :when-modified (cad-update))
    (Y_LENGTH
        :doc-string "Y length relative to origin."
        :when-modified (cad-update))
    (Z_LENGTH
        :doc-string "Z length relative to origin."
        :when-modified (cad-update))
)
```

Several items are of interest in the definition of the base CAD shape objects. First, the frame *CAD-OBJECT* inherits from *BB-OBJECT*, the root of all frames within ACE. In this way, the *CAD-OBJECT* gets all the base functionality of ACE associated with *BB-OBJECT*. Second, the *CAD-OBJECT* slots *CAD_COLOR* and *CAD_LAYER* are used in the setup of the CAD system color and layer before the object is displayed. Third, the *CAD-OBJECT* slots *ORIGIN* and *ROTATION* have a when-modified function attached. This when-modified function, *cad-update-linked*, will call the *cad-draw* handler when these slot values are modified. The function will also attempt to draw any linked objects in the CAD system following the *HAS-PART* semantic link. The code for this when-modified function is:

```
(defun cad-update-linked (inst slot old new)
    (if (send-msg inst :get-view-tag) (send-msg inst :cad-draw))
    ;redraw all connected items if drawable
    (if (send-msg inst :get-drawable)
        (mapcar
            #'(lambda (inst2) (cad-update-linked inst2 nil nil nil))
            (cad-linked-items (from-items inst)))))
```

In the *3D-CUBE-CAD-SHAPE-OBJECT* frame definition, note the when-modified function on the *X_LENGTH*, *Y_LENGTH*, and *Z_LENGTH* slots. This when-modified function, *cad-update*, will call the *cad-draw* handler when any slot's values are modified but will not attempt to draw any linked objects, unlike the when-modified function

*cad-update-linked*. This difference is because these slot values relate only to the view or dimension of this object. The *ORIGIN* and *ROTATION* slots for *CAD-OBJECT* relate to actual position of the object. A change of position may require movement of linked objects. The code for this when-modified function is:

```
(defun cad-update (inst slot old new)
    (if (send-msg inst :get-view-tag) (send-msg inst :cad-draw)))
```

The frame definition for a *DOOR* that inherits from the *3D-CUBE-CAD-SHAPE-OBJECT* is:

```
(define-frame DOOR
    (:is (3D-CUBE-CAD-SHAPE-OBJECT BUILDING COMPONENT))
    (HAND :constraints (:one-of (left hand right hand))
        :default-values (left hand))
    (MATERIAL :constraints (:one-of (wood metal))
        :default-values (wood))
    (HEIGHT :default-values (2100)
        :unit mm
        :map-to-slot z length
        :when-modified (map-to))
    (WIDTH :default-values (900)
        :unit mm
        :map-to-slot x length
        :when-modified (map-to))
    (DEPTH :default-values (140)
        :unit mm
        :map-to-slot y length
        :when-modified (map-to))
    (CAD LAYER :default-values (DOOR))
)
```

To associate base CAD shape slots with slots in the *DOOR* definition, note the *map-to-slot* attribute of the *HEIGHT*, *WIDTH*, and *DEPTH* slots of the *DOOR* frame definition. This attribute is used by the when-modified function *map-to* associated with each of these slots. When either the *HEIGHT*, *WIDTH*, or *DEPTH* slot values of a *DOOR* object are modified, the *map-to* when-modified function will run. This function will retrieve the slot named in the *map-to-slot* attribute and modify the corresponding mapped-to slot. In this case, modifying the *HEIGHT* slot will cause the *Z LENGTH* slot value to be modified as well. Since the *Z LENGTH* slot value is modified, it will automatically call the *cad-update* when-modified function, which will in turn call *cad-*

*draw*. The *acad-draw* handler (the *cad-draw* handler specialized for AutoCAD) for the *3D-CUBE-CAD-SHAPE-OBJECT* frame is:

```
(define-handler (3D-CUBE-CAD-SHAPE-OBJECT acad-draw) ()
    (if (and    (slot-value self 'origin) (slot-value self 'rotation) (slot-value self 'x  length)
                (slot-value self 'y  length) (slot-value self 'z  length))
                (progn
                        (send-msg self :acad-cleanup)
                        (send-msg self :acad-build (append `(insert-3dcube-main
                                (quote ,(send-msg self :get-absolute-origin))
                                ,(slot-value self 'x  length)
                                ,(slot-value self 'y  length)
                                ,(slot-value self 'z  length)
                                ,(nth 2 (send-msg self :get-absolute-rotation)))))
                        t)))
```

The *acad-draw* handler first checks that the slots needed for display have values. Second, it does a cleanup that erases any existing graphical object in AutoCAD. (A new object is drawn, not a modification of an existing object.) Finally, it displays the graphical object in AutoCAD using the needed slots from the symbolic object. Through this inheritance, instances created from the *DOOR* frame definition can be automatically represented graphically within the CAD system.

### *Create a Custom Drawing Function*

This approach requires the creation of a custom drawing function for each CAD-aware symbolic object. In addition, implementation of the when-modified capability for frame slots required for display must be handled. For instance, a *ROOF* frame definition and its *acad-draw* handler is defined below:

```
(define-frame ROOF
    (:is BUILDING  COMPONENT)
    (CORNERS
            :unit mm
            :when-modified (cad-update))
    (FACIA  DEPTH :default-values (300)
            :unit mm
            :when-modified (cad-update))
    (ORIENTATION :constraints (:one-of (north-south east-west))
            :default-values (north-south)
            :when-modified (cad-update))
```

```
    (OVERHANG :default-values (600)
              :unit mm
              :when-modified (cad-update))
    (PITCH :default-values (0.25)
              :unit %
              :when-modified (cad-update))
    (DECK :constraints (:one-of (wood concrete steel))
              :default-values (wood))
    (INSULATION :constraints (:one-of (none glass  fiber polyisocyanurate
                        polystyrene polyurethane))
              :default-values (glass  fiber))
    (ROOF  SYSTEM :constraints (:one-of (steep  slope low  slope))
              :default-values (low  slope))
    (TYPE :constraints (:one-of (gable hip flat))
              :default-values (flat)
              :when-modified (cad-update))
)

(define-handler (ROOF acad-draw) ()
    (send-msg self :acad-cleanup)
    (send-msg self :build-component `(eval ,(append '(make-3d-roof)
              (setup-slots self '(corners orientation pitch type facia  depth overhang)))))
    t)
```

Relevant slots have an associated when-modified function *cad-update* (as described on p 20) that will call the *acad-draw* handler when any slot values are modified. Note that the frame does not inherit from any predefined CAD shape. The *acad-draw* handler calls an AutoCAD function *make-3d-roof* that uses the *CORNERS*, *ORIENTATION*, *PITCH*, *TYPE*, *FACIA DEPTH*, and *OVERHANG* slots of the *ROOF* frame. This function was specifically written in AutoCAD for the display of a *ROOF* object. To create a custom graphical representation, a developer would have to write native CAD functions for each CAD system supported, as well as the code required within ACE.

## System Startup

To start the interaction between ACE and CAD, only one function needs to be called from within Lisp: *(go-to-cad 'user)*. This command will determine the current CAD system (set in the project information of ACE) and start the CAD system, which will then start CBrain in both ACE and AutoCAD, thus establishing communication. The "user" symbol is the name of the agent to which instances created within AutoCAD will be assigned.

In addition to starting the CAD system and establishing communication, this function can set up the user interface for the CAD system. Before calling the *go-to-cad* function, the function *cad-setup-command-add* can be called. This function accepts one argument, a command that AutoCAD will execute after communication is established but before control is returned to the user. This command could load a specific AutoCAD menu or some other agent specific AutoCAD setup feature. Once the default interaction between ACE and AutoCAD is set up, the systems will look as shown in Figure 5. (Notice the 'CADTalk' menu option in AutoCAD.)

After initialization, the user can create ACE symbolic objects from within AutoCAD. These objects will be displayed in AutoCAD using a set of predefined functions written for AutoCAD. Figure 6 shows an example of the default user interface created for AutoCAD for ACE symbolic object creation. An item can be selected for creation from the list of objects displayed. In this example, a *WALL* object is to be created. The *WALL* object inherits from the *3D-CUBE-CAD-SHAPE-OBJECT*. During creation, the *cad-draw* handler for the *WALL* frame is called, and its graphical representation is created in the AutoCAD drawing. The definition for the *cad-draw* handler, and specifically the *acad-draw* handler in this case, is the same as described on p 21.

With a new instance created, a user can select from the menu to display symbolic information from ACE about this object. The CAD system will request the user to

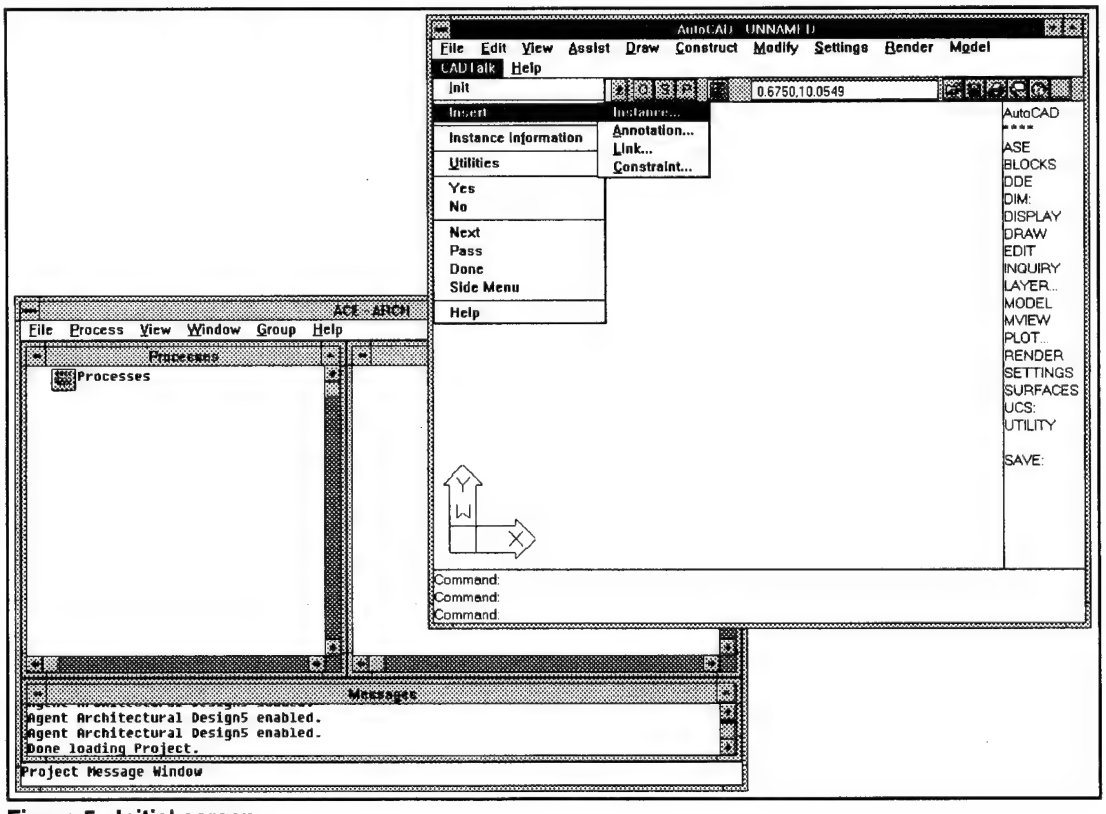


**Figure 5. Initial screen.**

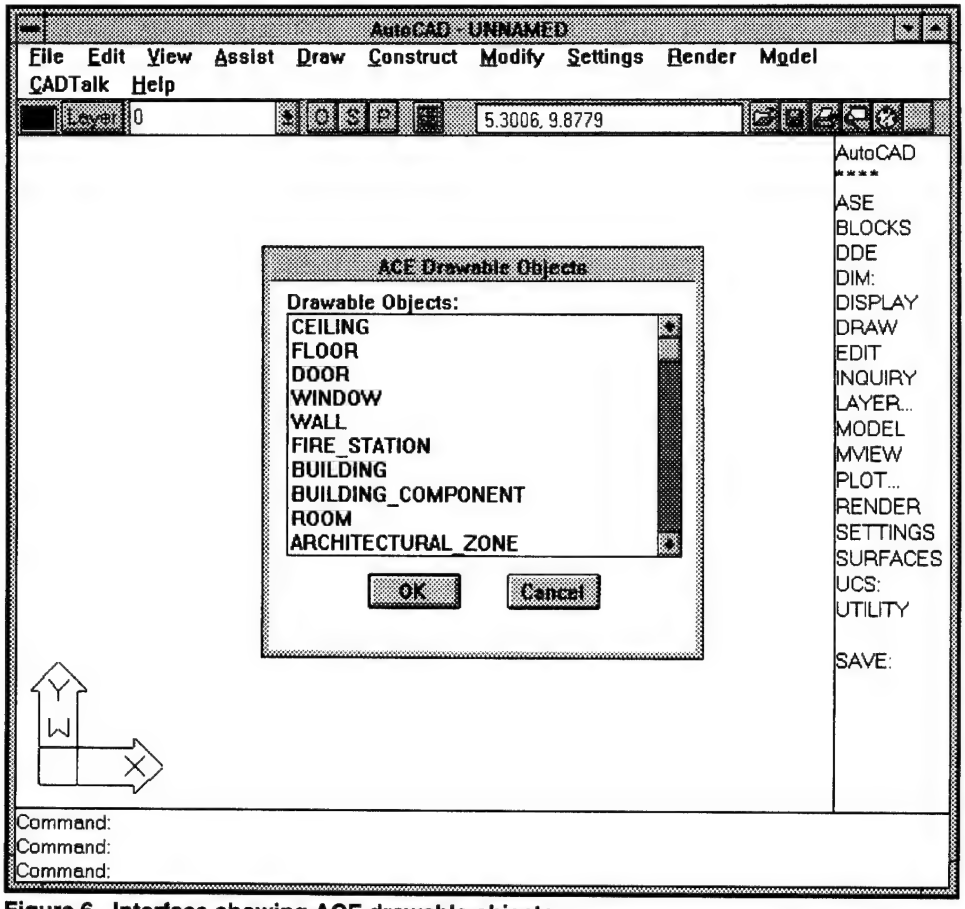


**Figure 6. Interface showing ACE drawable objects.**

select an object, a request will be made to ACE for slot values, and results will display in a predefined AutoCAD dialog box (see Figure 7).

## Display of Symbolic Objects

Once a CAD-aware frame definition has been created, drawing all instances of a particular frame is possible with a call to the function *cad-draw-all*. This function, listed below, takes a frame name as an argument and then runs the *cad-draw* handler on each frame instance.

```
(defun cad-draw-all (frame)
    (if (stringp frame) (setq frame (read-from-string frame)))
    (mapcar #'(lambda (inst) (send-msg inst :cad-draw)) (frame-instances frame))
    t)
```

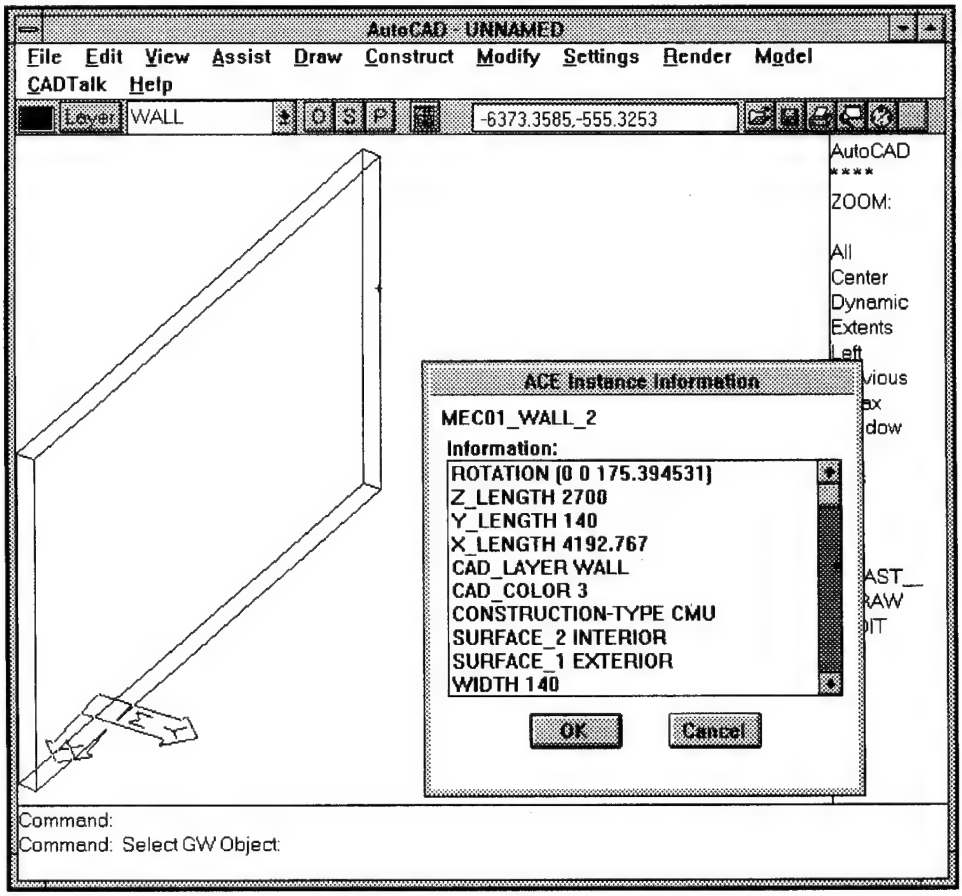


**Figure 7. AutoCAD showing ACE symbolic information.**

From within AutoCAD, this command can be sent to ACE with the statement *'(cl-eval '(cad-draw-all 'wall))'*, for example, to draw all ACE *WALL* instances in the AutoCAD system. Figure 8 is an example of an ACE project and its corresponding AutoCAD drawing.

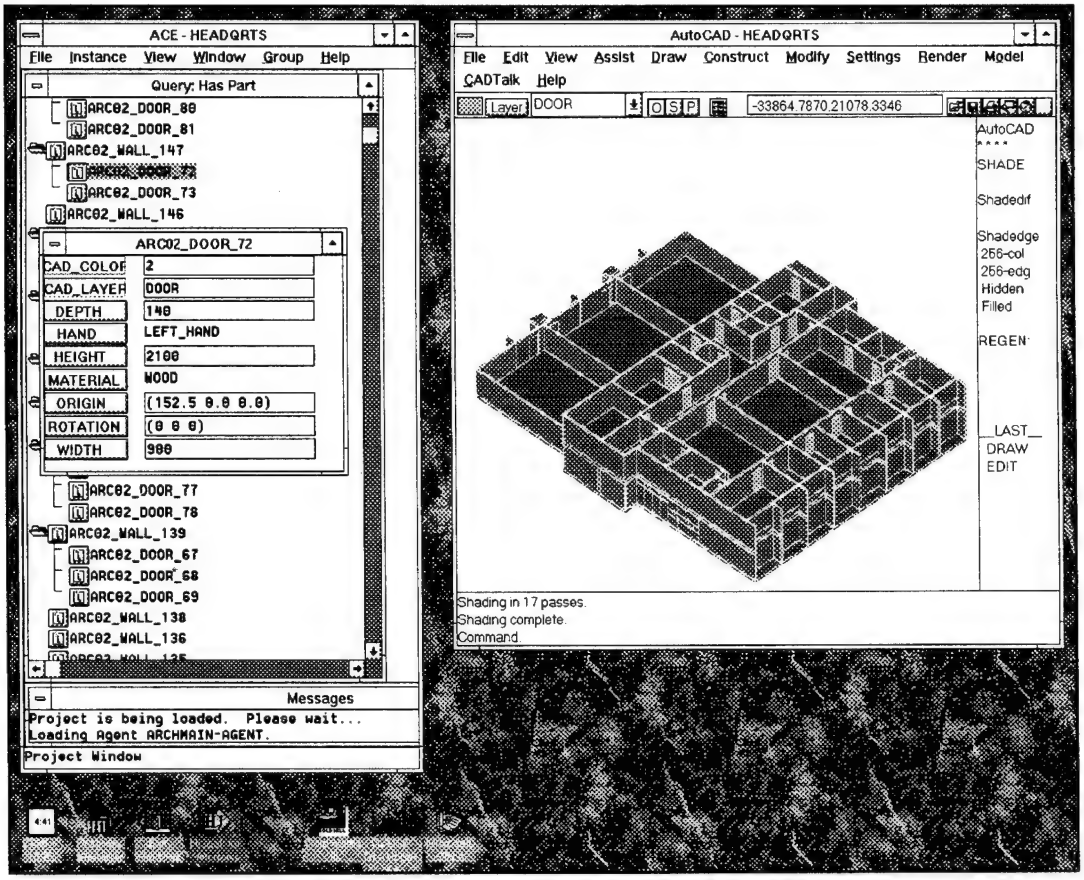


**Figure 8. Larger example of an ACE project and corresponding AutoCAD drawing.**

# 8 Conclusions

A methodology (CBrain) to allow efficient and reliable bidirectional communication between the ACE and CAD systems now enables objects within ACE to be represented and manipulated within the CAD system. A methodology for linking symbolic objects in ACE with graphical objects in CAD was described and implemented as the CADTalk system within ACE.

The CBrain application has proved to be a reliable communication interface. The ability to pack strings together into one DDE message and allow recursive calling between ACE and CAD has worked well. The DDE message passing is reliable, and the speed is fairly good. The use of Dynamic Link Libraries (DLLs) or Object Linking and Embedding (OLE) would probably increase the communication speed, but would be hampered by the inability to program within the commercial CAD systems, whose code is proprietary. The advantage of the DDE technology is that it is established in industry and works on the Windows®, Windows NT®, and Windows 95® platforms. The functions written, *ac-eval* and *cl-eval*, are low-level basic functions that can be used by developers to create more complex functions.

The CADTalk methodology provides great flexibility to agent developers. The ability to establish relations between objects, inherit from base CAD shapes, and create custom CAD graphical representations allow tremendous flexibility to the agent developer. Through the inheritance of base CAD shape objects, an agent developer can avoid much of the coding required to make CAD-aware objects. Coding is still required if the developer is to make frames with complex actions or representations within CAD. If a user wishes to develop more complex objects, some extra effort and knowledge is required. Knowledge of AutoLisp® and how to perform certain actions within the CAD system is necessary as this approach can be fairly complex. A custom user interface can be added to the CAD system, but, again, more effort and knowledge is required.

Future research will seek to use Microsoft® Windows® technologies to improve the speed of the CBrain communication interface. These technologies include the use of DLL and OLE mechanisms.

# References

*GCLisp Developer, Reference Manual* (Gold Hill, Inc., 1990).

Hoff, Kendra Z., Sara E. Ort, Bruce L. Rives, and Kirk D. McGraw, *ACE 1.1 User's Manual*, ADP Report 95/54 (U.S. Army Construction Engineering Research Laboratory, June 1995).

Steele, Guy L. Jr., *Common Lisp, The Language, Second Edition* (Digital Press, 1990).

# Appendix A: Source Code Listing

This appendix shows actual portions of code from files used during program execution, including the internal details of the API listed in Appendix B and utility functions implemented.

## ACE Source Code

File: cad01.lsp
Contains system startup functions.

```
;;
;;************************** GENERIC SETUP/STARTUP ROUTINES **********************
;;

;;*******************************************************************************
;Go to CAD, agent passed is string of agent starting CAD.
;For now: (slot-value 'project-info 'cad-system) must equal AUTOCAD-WIN-12 or
MICROSTATION-NT!
(defun go-to-cad (agent &aux cad-system command-line)
;Check CAD system variables.
  (when (check-cad-variables)
;Get some values needed.
      (setq cad-system (current-cad-system))
      (setq command-line (slot-value 'project-info 'cad-command-line))
;Setup CBrain.
      (cbrain-debug t) ;remove after debugged
      (cbrain-in-package "GW")
;Setup global variables.
      (setq dde::*current-agent* agent)
      (setq dde::*current-interp* nil)
;Startup CAD system.
      (if (eql 'AUTOCAD-WIN-12 cad-system)
        (acad-startup cad-system command-line)
        (mstation-startup cad-system command-line))
;Start CBrain.
      (cbrain-start)))

;;*******************************************************************************
(defun check-cad-variables (&aux result system)
  (setq result t)
  (setq system (current-cad-system))
  (when (not (frame-p system))
      (text (format nil "CAD System Incorrect: ~A, Change in File/Properties." system))
      (setq result nil))
  (when (not (member system '(AUTOCAD-WIN-12 MICROSTATION-NT)))
      (text (format nil "CAD System ~A Incorrect: Change in File/Properties." system))
```

```
      (setq result nil))
  (when (not (probe-file (slot-value 'project-info 'cad-command-line)))
      (text (format nil
        "CAD Command Line Incorrect, Run File Not Found: ~A, Change in File/Properties."
        (slot-value 'project-info 'cad-command-line)))
      (setq result nil))
  (when (not (directoryp (slot-value 'project-info 'cad-drawing-directory)))
      (text (format nil
         "CAD Drawing Directory Incorrect, Path File Not Found: ~A, Change in File/
Properties."
        (slot-value 'project-info 'cad-drawing-directory)))
      (setq result nil))
  result)


;;*****************************************************************************
;Find out if AutoCAD/Microstation is already running.
(defun cad-running ()
  (if (eql 'AUTOCAD-WIN-12 (current-cad-system))
;Autocad
      (let ((cad-chnl (dde:channel-initiate "AutoCAD.DDE" "system")))
        (if cad-chnl (progn (dde:channel-terminate cad-chnl) T) NIL))
;Microstation
      (let ((cad-chnl (dde:channel-initiate "ustn" "keyin")))
        (if cad-chnl (progn (dde:channel-terminate cad-chnl) T) NIL))))

;;*****************************************************************************
;Setup AutoCAD/Microstation that is already running for CADTalk.
(defun setup-running-cad ()
  (if (eql 'AUTOCAD-WIN-12 (current-cad-system))
;Autocad, have it load ctinit.lsp and run CADTALK command.
      (let ((cad-chnl (dde:channel-initiate "AutoCAD.DDE" "system")))
      (when cad-chnl
        (dde:channel-execute cad-chnl
          (format nil "[(load ~S) CADTALK (init) ]" (build-local "acad.lsp")))
        (dde:channel-terminate cad-chnl)))
;Microstation, have it load ddeclkcn and do cbrain_start.
      (let ((cad-chnl (dde:channel-initiate "ustn" "keyin")))
      (when cad-chnl
        (dde:channel-execute cad-chnl "mdl load mscbrain")
        (dde:channel-execute cad-chnl "mdl load cadcmds")
        (dde:channel-execute cad-chnl "cbrain_start")
        (dde:channel-terminate cad-chnl)))
))

;;*****************************************************************************
(setq *cad-setup-commands* ())
;; Add a CAD setup command.
(defun cad-setup-command-add (command)
      (setq *cad-setup-commands* (reverse (append (list command) (reverse
      *cad-setup-commands*)))))

;;*****************************************************************************
;; Run the CAD setup commands, called by AutoCAD during initialization.
(defun cad-setup-command-run ()
  (when *cad-setup-commands*
      (mapcar #'(lambda (command) (ac-eval-text command)) *cad-setup-commands*))
  (cad-setup-command-clear))

;;*****************************************************************************
;; Clear the CAD setup command list.
```

```
(defun cad-setup-command-clear ()
  (setq *cad-setup-commands* ()))

;;
;;*************** ROUTINES TO SETUP AUTOCAD FILES FOR EXCURSION *****************
;;

;;*******************************************************************************
(defun acad-startup (cad-system command-line &aux dr-obj)
  (autocad-write-start-lisp cad-system)
  (cond
      ((if (cad-running) (setup-running-cad)))
      ((when (setq dr-obj (select-drawobj)) (autocad-run-self command-line dr-obj)))
      (t (autocad-run-self command-line nil))))

;;*******************************************************************************
;; Write the file containing the AUTOLISP code which is run when CADTALK starts up.
(defun autocad-write-start-lisp (frame &aux cadtalk-dir)
  (setq cadtalk-dir (frame-slot-default frame 'cadtalk-directory))
  (with-open-file (file (build-local "acad.lsp") :direction :output)
      (format file
"(princ \"\\nReading ACE acad.lsp.\\n\")
~%")
      (format file
"(defun C:CADTALK ()
(vmon)
(setvar \"CMDECHO\" 0)
(setq *windows-acad* ~S)
(command \"MENU\" ~S)
(princ \"Loading Cadtalk...\")
(setq *ace-local* ~S)
(setq *ace-net* ~S)
(setq *cadtalk-directory* ~S)
(setq acad-requests ~S)
(setq acad-results  ~S)
(setq donefile      ~S)
(load ~S)
(load ~S)
(load ~S)
(princ \"loaded.\")
*nothing*)
~%"
      (if (member :GCLISP *features*) t nil)
      (namestring (make-path (list cadtalk-dir) (frame-slot-default frame 'menu-file)))
      *ace-local*
      *ace-net*
      cadtalk-dir
      (build-local (frame-slot-default frame 'request-file))
      (build-local (frame-slot-default frame 'result-file))
      (build-local (pathname-no-ext (frame-slot-default frame 'done-script)))
      (namestring (make-cad-mod-path (list "acad") "acadbase.lsp"))
      (namestring (make-cad-mod-path (list "acad") "acadutil.lsp"))
      (namestring  (make-path  (list  cadtalk-dir)  (frame-slot-default  frame  'con-
trol-file))))
      (format file
"(defun S::STARTUP ()
(C:CADTALK)
(init))
~%")
))
```

```
;;*****************************************************************************
;; command-line is a string, ex: "d:/acad12w/acad.exe".
;; drawing is a string.
(defun autocad-run-self (command-line &optional drawing)
  (let* ((dir (cd))) ; save the current directory
      (cd *ace-local*) ; change to local directory (where acad.lsp made)
      (cond
        ((null drawing) (sys:exec command-line ""))
        (t (sys:exec (format nil "~A ~A" command-line drawing) "")))
      (cd dir))) ; restore directory

;;
;;*************** ROUTINES TO SETUP MICROSTATION FILES FOR EXCURSION ************
;;

;;*****************************************************************************
(defun mstation-startup (cad-system command-line &aux dr-obj)
  (if (not (cad-running))
      (mstation-run-self command-line)
      (setup-running-cad)))

;;*****************************************************************************
;; command-line is a string, ex: "d:/acad12w/acad.exe".
;; drawing and start-script are strings.
(defun mstation-run-self (command-line)
  (let* ((dir (cd)) ; save the current directory
          (cad-dir (pathname-device-dir command-line))) ; device and directory of CAD
executable
         (cd cad-dir) ; change to the autocad directory
#+:GCLISP (sys:exec command-line "")
      (cd dir))) ; restore directory
```

## File: cadbb.lsp

Contains low-level *BB-OBJECT* handlers (later specilized for use).

```
(setf *cad-shape-objects*
  '(CAD-OBJECT CAD-SHAPE-OBJECT LINE-CAD-SHAPE-OBJECT
    2D-POLYGON-CAD-SHAPE-OBJECT  3D-POLYGON-CAD-SHAPE-OBJECT  3D-CUBE-CAD-SHAPE-OBJECT
3DFACE-CAD-SHAPE-OBJECT))

;;---------------------------------------------------------------------------
;;---------------------------------------------------------------------------
;; DEFINE A FACET FOR USE WITH DRAWABLE OBJECTS.
;;---------------------------------------------------------------------------
;;---------------------------------------------------------------------------

;*****************************************************************************
(define-facet :unit)

;*****************************************************************************
(define-handler (BB-OBJECT get-slot-unit) (slot)
  (slot-facet self slot :unit))

;*****************************************************************************
(define-handler (BB-OBJECT set-slot-unit) (slot new)
  (setf (slot-facet self slot :unit) new))

;;---------------------------------------------------------------------------
;;---------------------------------------------------------------------------
```

```
;; FUNCTIONS CALLED FROM CAD TO TEST/PROCESS DRAWABLE OBJECTS.
;;---------------------------------------------------------------------------
;;---------------------------------------------------------------------------

;***************************************************************************
;Routine to test if a frame is drawable.
(defun is-drawable-frame (frame)
  (if (stringp frame) (setq frame (read-from-string frame)))
  (car (slot-facet frame 'DRAWABLE :default-values)))

;***************************************************************************
;Routine to return all drawable frames, returns a list of strings.
(defun all-drawable ()
  (let ((items (mapcar 'gw-name (frame-all-children 'BB-OBJECT))))
       (setf items (set-difference items *cad-shape-objects*))
       (setf items (mapcan #'(lambda (name) (if (is-drawable-frame name) (list name)
())) items))
       (mapcar 'symbol-name items)))

;***************************************************************************
;Routine to do main amount of work in trying to create a drawable object.
(defun process-drawable-object (frame agent)
  (if (stringp frame) (setq frame (read-from-string frame)))
  (if (stringp agent) (setq agent (read-from-string agent)))
  (if (is-drawable-frame frame)
       (let ((inst (ace-instance frame)))
         (if (and (send-msg inst :cad-create) (send-msg inst :cad-draw))
           t
           (progn (send-msg inst :delete) nil)))
       (progn (print "Frame Is Not Drawable!") nil)))

;***************************************************************************
;Routine to draw all instances of the given frame.
(defun cad-draw-all (frame)
  (if (stringp frame) (setq frame (read-from-string frame)))
  (mapcar #'(lambda (inst) (send-msg inst :cad-draw)) (frame-instances frame))
  t)

;;---------------------------------------------------------------------------
;;---------------------------------------------------------------------------
;; GENERIC HANDLERS
;;---------------------------------------------------------------------------
;;---------------------------------------------------------------------------

;***************************************************************************
(define-handler (BB-OBJECT get-drawable) ()
  (slot-value self 'drawable))

;***************************************************************************
(define-handler (BB-OBJECT set-drawable) (new)
  (setf (slot-value self 'drawable) new))

;***************************************************************************
;Routine to delete a drawable object.
(define-handler (BB-OBJECT delete-drawable-object) ()
  (send-msg self :delete))

;***************************************************************************
;Routine to set the VIEWS slot.  NOTE: tags is a list of tags, which are strings!
(define-handler (BB-OBJECT set-view-tag) (tags)
  (if (eql 'AUTOCAD-WIN-12 (current-cad-system))
```

```
        (send-msg self :set-entity-handle tags)
        (send-msg self :set-ms-element-tag tags)))

;*****************************************************************************************
;Routine to get the VIEWS slot. NOTE: returns a list of tags, which are strings!
(define-handler (BB-OBJECT get-view-tag) ()
  (if (eql 'AUTOCAD-WIN-12 (current-cad-system))
        (send-msg self :get-entity-handle)
        (send-msg self :get-ms-element-tag)))

;*****************************************************************************************
;Routine to clear the VIEWS slot.
(define-handler (BB-OBJECT clear-view-tag) ()
  (if (eql 'AUTOCAD-WIN-12 (current-cad-system))
        (send-msg self :clear-entity-handle)
        (send-msg self :clear-ms-element-tag)))

;*****************************************************************************************
;Routine to set the instance name associated with the tags, name is a string.
(define-handler (BB-OBJECT set-cad-inst-name) (name)
  (if (eql 'AUTOCAD-WIN-12 (current-cad-system))
        (send-msg self :acad-add-xdata-all name)
        (send-msg self :ms-add-xdata-all name)))

;*****************************************************************************************
;Routine to get the instance name associated with the tags.
(define-handler (BB-OBJECT get-cad-inst-name) ()
  (if (eql 'AUTOCAD-WIN-12 (current-cad-system))
        (send-msg self :acad-get-xdata)
        (send-msg self :ms-get-xdata)))

;*****************************************************************************************
;Called to get initial slot values from CAD system before cad-draw.
(define-handler (BB-OBJECT cad-create) ()
  (if (send-msg self :get-drawable)
        (if (eql 'AUTOCAD-WIN-12 (current-cad-system))
          (send-msg self :acad-create)
          (send-msg self :mstation-create))))

;*****************************************************************************************
;Draw object.
(define-handler (BB-OBJECT cad-draw) ()
  (if (send-msg self :get-drawable)
        (if (eql 'AUTOCAD-WIN-12 (current-cad-system))
          (send-msg self :acad-draw)
          (send-msg self :mstation-draw))))

;*****************************************************************************************
;Routine to get editable slots.
(define-handler (BB-OBJECT editable-slots) ()
  (mapcar #'(lambda (slot) (list slot (type-of (slot-value self slot))))
        (set-difference (instance-all-slots self) (frame-slots 'bb-object))))

;;---------------------------------------------------------------------------------------
;;---------------------------------------------------------------------------------------
;; AUTOCAD SPECIFIC HANDLERS
;;---------------------------------------------------------------------------------------
;;---------------------------------------------------------------------------------------

;*****************************************************************************************
```

```
;Routine to set the entity handle within the VIEWS slot.  NOTE: handle is a list of
entity handles, which are strings!
(define-handler (BB-OBJECT set-entity-handle) (handle)
  (if (listp handle)
        (let ((item (assoc 'ACAD (slot-value self 'views))))
          (if item
                (setf (slot-value self 'views) (subst (cons 'ACAD (list handle)) item
(slot-value self 'views)))
              (setf (slot-value self 'views) (acons 'ACAD (list handle) (slot-value self
'views)))))
        (print "Incorrect handle to set-entity-handle, must be a list!")))

;*****************************************************************************************
;Routine to get the entity handle within the VIEWS slot. NOTE: returns a list of entity
handles, which are strings!
(define-handler (BB-OBJECT get-entity-handle) ()
  (cadr (assoc 'ACAD (slot-value self 'views))))

;*****************************************************************************************
;Routine to clear the entity handle within the VIEWS slot.
(define-handler (BB-OBJECT clear-entity-handle) ()
   (setf (slot-value self 'views) (remove (assoc 'ACAD (slot-value self 'views))
(slot-value self 'views))))

;*****************************************************************************************
;Handler to return instance name for each tag in list, returns list.
(define-handler (BB-OBJECT acad-get-xdata) ()
   (mapcar #'(lambda (x) (ac-eval `(cdr (entget (handent ,x))))) (send-msg self
:get-view-tag)))

;*****************************************************************************************
;Handler to add instance name to CAD object, data is string.
(define-handler (BB-OBJECT acad-add-xdata-all) (data)
  (dolist (handle (send-msg self :get-view-tag)) (send-msg self :acad-add-xdata-one
handle data)))

;*****************************************************************************************
;Handler to add instance name to CAD object with the given ID, ID and data are strings.
(define-handler (BB-OBJECT acad-add-xdata-one) (ID data)
  (ac-eval `(add-xdata ,ID ,data)))

;*****************************************************************************************
;Handler to set initial slot values before cad-draw.
;Return t if successful, else ().
(define-handler (BB-OBJECT acad-create) () t)

;*****************************************************************************************
;Handler responsible for drawing object, setting entity handle, and linking to object in
CAD.
;Return t if successful, else ().
(define-handler (BB-OBJECT acad-draw) (&aux handle)
  (if (setq handle (ac-eval `(insert-text nil ,(symbol-name (gw-name self)))))
        (progn
          (send-msg self :set-view-tag handle)
          (send-msg self :acad-add-xdata-all (symbol-name (gw-name self)))
          t)
        (progn (print "Error drawing object.") ())))

;*****************************************************************************************
;Handler to act when object was moved in AutoCad.
(define-handler (BB-OBJECT acad-object-moved) (offset)
```

```
  (print (format nil "Instance ~S moved in AutoCad ~S." (gw-name self) offset)))

;*****************************************************************************************
;Handler to act when object was erased in AutoCad.
(define-handler (BB-OBJECT acad-object-erased) ()
  (print (format nil "Instance ~S erased in AutoCad." (gw-name self))))

;*****************************************************************************************
;Handler to act when object was copied in AutoCad.
(define-handler (BB-OBJECT acad-object-copied) ()
  (print (format nil "Instance ~S copied in AutoCad." (gw-name self))))

;*****************************************************************************************
;Handler to highlight an object in AutoCad.
(define-handler (BB-OBJECT acad-highlight) ()
   (dolist (handle (send-msg self :get-view-tag)) (ac-eval `(redraw (handent ,handle)
3))))

;*****************************************************************************************
;Handler to highlight an object in AutoCad.
(define-handler (BB-OBJECT acad-unhighlight) ()
   (dolist (handle (send-msg self :get-view-tag)) (ac-eval `(redraw (handent ,handle)
4))))

;*****************************************************************************************
;Handler to change the color of an object in AutoCad.
;c is an integer between 0 and 255 (both inclusive).
(define-handler (BB-OBJECT set-color) (c)
  (ac-eval `(set-color (quote ,(send-msg self :get-view-tag)) ,c)))

;*****************************************************************************************
;Handler to get the color of an object in AutoCad.
;returns a list of color numbers or nil depending on the colors assigned to each of the
handles in the list.
(define-handler (BB-OBJECT get-color) ()
  (ac-eval `(get-color (quote ,(send-msg self :get-view-tag)))))

;;---------------------------------------------------------------------------------------
;;---------------------------------------------------------------------------------------
;; MICROSTATION SPECIFIC HANDLERS
;;---------------------------------------------------------------------------------------
;;---------------------------------------------------------------------------------------

;routine to turn a list of symbols into a list of strings
(defun make-list-string (lst)
  (apply 'string-append (mapcar #'(lambda (x) (string-append (write-to-string x) " "))
(coerce lst 'list))))

;routine to turn a list of lists into a list of strings
(defun list-make-list-string (lst &aux temp temp2)
  (setq temp (mapcar 'make-list-string lst) temp2 "")
  (dolist (x temp) (setq temp2 (string-append temp2 x)))
  temp2)

;*****************************************************************************************
;Routine to set the element tag within the VIEWS slot.  NOTE: tag is a list.
(define-handler (BB-OBJECT set-ms-element-tag) (tag)
  (if (listp tag)
        (let ((item (assoc 'USTN (slot-value self 'views))))
          (if item
```

```
                (setf (slot-value self 'views) (subst (cons 'USTN (list tag)) item
(slot-value self 'views)))
              (setf (slot-value self 'views) (acons 'USTN (list tag) (slot-value self
'views)))))
        (print "Incorrect tag to set-ms-element-tag, must be a list!")))

;*****************************************************************************************
;Routine to get the element tag within the VIEWS slot. NOTE: returns a list of tags,
which are numbers!
(define-handler (BB-OBJECT get-ms-element-tag) ()
  (cadr (assoc 'USTN (slot-value self 'views))))

;*****************************************************************************************
;Routine to clear the element tag within the VIEWS slot.
(define-handler (BB-OBJECT clear-ms-element-tag) ()
   (setf (slot-value self 'views) (remove (assoc 'USTN (slot-value self 'views))
(slot-value self 'views))))

;*****************************************************************************************
;Handler to return instance name for each tag in list, returns list.
(define-handler (BB-OBJECT ms-get-xdata) ()
  (mapcar #'(lambda (x) (ac-eval-text (format NIL "get_xdata ~A" x))) (send-msg self
:get-view-tag)))

;*****************************************************************************************
;Handler to add instance name to CAD object, data is string.
(define-handler (BB-OBJECT ms-add-xdata-all) (data)
  (dolist (handle (send-msg self :get-view-tag)) (send-msg self :ms-add-xdata-one handle
data)))

;*****************************************************************************************
;Handler to add instance name to CAD object with the given ID, ID and data are strings.
(define-handler (BB-OBJECT ms-add-xdata-one) (ID data)
  (ac-eval-text (format NIL "set_xdata ~A ~A" ID data)))

;*****************************************************************************************
;Handler to set initial slot values before cad-draw.
;Return t if successful, else ().
(define-handler (BB-OBJECT mstation-create) () t)

;*****************************************************************************************
;Handler responsible for drawing object, setting entity handle, and linking to object in
CAD.
;Return t if successful, else ().
(define-handler (BB-OBJECT mstation-draw) (&aux tag)
  (if (setq tag (ac-eval-text (format NIL "insert_text 0 0 0 ~S ~S" (gw-name self)
(gw-name self))))
    (progn
      (send-msg self :set-view-tag tag)
      t)
    (progn (print "Error drawing object.") ())))
```

## File: cadshape.lsp

Contains base CAD shape definitions, file added to ACE library for use.

```
;;-----------------------------------------------------------------------------
;;-----------------------------------------------------------------------------
;; DEFINE BASE CAD-OBJECT
;;-----------------------------------------------------------------------------
```

```
;;-------------------------------------------------------------------------------

(DEFINE-FRAME CAD-OBJECT
   (:is BB-OBJECT)
   (DRAWABLE :default-values (t))
   (ORIGIN :default-values ((0 0 0))
        :when-modified (cad-update-linked))
   (ROTATION :default-values ((0 0 0))
        :when-modified (cad-update-linked))
   (CAD_COLOR :default-values (1) :no-inference T)
   (CAD_LAYER :default-values (0) :no-inference T)
)

;;-------------------------------------------------------------------------------
;;-------------------------------------------------------------------------------
;; LINKED ITEM FUNCTIONS
;;-------------------------------------------------------------------------------
;;-------------------------------------------------------------------------------

;routine to return a list of items that are connected to the given inst
(defun to-items (inst)
  (mapcar #'(lambda (slink) (list (slink-from slink) (slink-type slink)))
    (send-msg inst :get-slinks-to)))

;routine to return a list of items that are connected from the given inst
(defun from-items (inst)
  (mapcar #'(lambda (slink) (list (slink-to slink) (slink-type slink)))
    (send-msg inst :get-slinks-from)))

;routine to return a list of items with has-part links in the given link-list;
;link-list comes from to-parts or from-parts above (ex: (has-part-items (to-items
'west_wall)))
(defun has-part-items (link-list)
  (let ((result ()))
    (mapcar
      #'(lambda (slink) (if (eql 'has-part (cadr slink))
        (setf result (append (list (car slink)) result)) result))
      link-list)
    result))

;routine to return a list of items with cad links in the given link-list;
;link-list comes from to-parts or from-parts above  (ex: (cad-linked-items (to-items
'west_wall)))
;(NOTE: USING HAS-PART AS THE CAD-LINK!)
(defun cad-linked-items (link-list)
  (let ((result ()))
    (mapcar
      #'(lambda (slink) (if (eql 'has-part (cadr slink))
        (setf result (append (list (car slink)) result)) result))
      link-list)
    result))

;;-------------------------------------------------------------------------------
;;-------------------------------------------------------------------------------
;; WHEN MODIFIED FUNCTIONS
;;-------------------------------------------------------------------------------
;;-------------------------------------------------------------------------------

(setf *cad-update* t)
(setf *cad-update-items* ())
```

```
(defun cad-update-save-inst (inst)
  (if (not (member inst *cad-update-items*))
    (setf *cad-update-items* (append (list inst) *cad-update-items*))))

;when-modified routine - USE BY CREATING MODIFIED HANDLER!
(defun slot-modified (inst slot old new)
  (send-msg inst :modified inst slot old new))

;when-modified routine - DOES NOT REDRAW CAD-LINKED-ITEMS!
(defun cad-update (inst slot old new)
  (declare (ignore slot new))
  (if *cad-update*
    (if (send-msg inst :get-view-tag) (send-msg inst :cad-draw))
    (cad-update-save-inst inst)))

;when-modified routine - DOES REDRAW CAD-LINKED-ITEMS!
(defun cad-update-linked (inst slot old new)
  (declare (ignore slot new))
  (if *cad-update*
    (progn
      (if (send-msg inst :get-view-tag) (send-msg inst :cad-draw))
;redraw all connected items if drawable
      (if (send-msg inst :get-drawable)
              (mapcar #'(lambda (inst2) (cad-update-linked inst2 nil nil nil))
              (cad-linked-items (from-items inst)))))
    (cad-update-save-inst inst)))

;;-------------------------------------------------------------------------------
;;-------------------------------------------------------------------------------
;; GENERIC HANDLERS
;;-------------------------------------------------------------------------------
;;-------------------------------------------------------------------------------

;*******************************************************************************
;Routine to setup geometry settings.
(define-handler (CAD-OBJECT cad-setup) ()
  (if (eql 'AUTOCAD-WIN-12 (current-cad-system))
        (send-msg self :acad-cad-setup)
        (send-msg self :mstation-cad-setup)))

;*******************************************************************************
;Routine called when a slot is modified.
(define-handler (CAD-OBJECT modified) (inst slot old new) t)

;;-------------------------------------------------------------------------------
;;-------------------------------------------------------------------------------
;; AUTOCAD SPECIFIC HANDLERS
;;-------------------------------------------------------------------------------
;;-------------------------------------------------------------------------------

;*******************************************************************************
;Routine to setup layer and color before drawing.
(define-handler (CAD-OBJECT acad-cad-setup) ()
  (ac-eval `(progn
    (setvar "CMDECHO" 0)
    (command ".LAYER" "M" ,(princ-to-string (slot-value self 'cad_layer))
                      "C" ,(princ-to-string (slot-value self 'cad_color)) "" "")
    (setvar "CMDECHO" 1))))

;*******************************************************************************
;Routine to erase CAD object and clear view slot.
```

```
(define-handler (CAD-OBJECT acad-cleanup) (&aux handle)
  (when (setq handle (car (send-msg self :get-view-tag)))
        (ac-eval `(if (handent ,handle) (entdel (handent ,handle))))
        (send-msg self :clear-view-tag)))

;*****************************************************************************************
;Routine to speed-up creation process, does combination of
;creation routine, adding xdata, and setting of view tag in GW.
(define-handler (CAD-OBJECT acad-build) (creation &aux ent)
  (send-msg self :cad-setup)
   (if (setf ent (ac-eval `(progn ,creation (add-xdata (entlast) ,(format nil "~S"
(gw-name self)))
        (entlast))))
        (send-msg self :set-view-tag (list (string-left-trim '"H-" (string ent))))))

;*****************************************************************************************
;Handler responsible for drawing object, setting entity handle, and linking to object in
CAD.
;Return t if successful, else ().
;NOTE: ASSUMING THAT ORIGIN HAS BEEN SET!
(define-handler (CAD-OBJECT acad-draw) ()
  (if (slot-value self 'origin)
        (progn
          (send-msg self :acad-cleanup)
          (send-msg self :acad-build `(insert-text (quote ,(slot-value self 'origin))
               ,(symbol-name (gw-name self))))
          t)))

;*****************************************************************************************
(define-handler (CAD-OBJECT acad-object-moved) (offset)
  (print (format nil "~S moved in AutoCad ~S." (gw-name self) offset))
;update origin slot - when modified called
  (setf (slot-value self 'origin) (add-points (slot-value self 'origin) offset))
  t)

;*****************************************************************************************
(define-handler (CAD-OBJECT acad-object-erased) ()
  (print (format nil "Instance ~S erased in AutoCad." (gw-name self)))
  (send-msg self :delete))

;;-------------------------------------------------------------------------------------
;;-------------------------------------------------------------------------------------
;; MICROSTATION SPECIFIC HANDLERS
;;-------------------------------------------------------------------------------------
;;-------------------------------------------------------------------------------------

;*****************************************************************************************
;Routine to setup layer and color before drawing.
(define-handler (CAD-OBJECT mstation-cad-setup) ()
  (ac-eval-text (format NIL "LV=~S" (slot-value self 'cad_layer)))
  (ac-eval-text (format NIL "CO=~S" (slot-value self 'cad_color))))

;*****************************************************************************************
;Routine to erase CAD object and clear view slot.
(define-handler (CAD-OBJECT mstation-cleanup) (&aux handle)
  (when (setq handle (car (send-msg self :get-view-tag)))
        (ac-eval-text (format NIL "delete_gw_obj ~A" handle))
        (send-msg self :clear-view-tag)))

;*****************************************************************************************
;Routine to speed-up creation process, does combination of
```

```
;creation routine, adding xdata, and setting of view tag in GW.
(define-handler (CAD-OBJECT mstation-build) (creation &aux tag)
  (send-msg self :cad-setup)
  (if (setf tag (ac-eval-text creation))
    (if (setf tag (ac-eval 'get_tag))
      (send-msg self :set-view-tag (list tag)))))

;*****************************************************************************************
;Handler responsible for drawing object, setting entity handle, and linking to object in
CAD.
;Return t if successful, else ().
(define-handler (CAD-OBJECT mstation-draw) (&aux tag)
  (if (slot-value self 'origin)
        (progn
          (send-msg self :mstation-cleanup)
          (send-msg self :mstation-build (format NIL "insert_text ~A ~S ~S"
           (make-list-string (slot-value self 'origin)) (gw-name self) (gw-name self)))
          t)))

;;---------------------------------------------------------------------------------------
;;---------------------------------------------------------------------------------------
;; BELOW IS THE CODE FOR OBJECTS THAT WILL BASED ON THE CAD HEIRARCHY METHODOLOGY (LCS
MODEL)
;;---------------------------------------------------------------------------------------
;;---------------------------------------------------------------------------------------

;;---------------------------------------------------------------------------------------
;;---------------------------------------------------------------------------------------
;; DEFINE A FACET FOR USE WITH CAD-SHAPE-OBJECTS (FACET WILL BE USED WITH MAP-TO WHEN
MODIFIED)
;;---------------------------------------------------------------------------------------
;;---------------------------------------------------------------------------------------

;*****************************************************************************************
(define-facet :map-to-slot)

;*****************************************************************************************
(define-handler (CAD-SHAPE-OBJECT get-map-to-slot) (slot)
  (slot-facet self slot :map-to-slot))

;*****************************************************************************************
(define-handler (CAD-SHAPE-OBJECT set-map-to-slot) (slot new)
  (setf (slot-facet self slot :map-to-slot) new))

;*****************************************************************************************
;when-modified function that will be used to map a slot value to another slot value
(defun map-to (inst slot old new)
  (declare (ignore old))
  (setf (slot-value inst (send-msg inst :get-map-to-slot slot)) (car new)))

;;---------------------------------------------------------------------------------------
;;---------------------------------------------------------------------------------------
;; DEFINE BASIC CAD-SHAPE-OBJECT
;;---------------------------------------------------------------------------------------
;;---------------------------------------------------------------------------------------

(DEFINE-FRAME CAD-SHAPE-OBJECT
   (:is CAD-OBJECT)
)

;;---------------------------------------------------------------------------------------
```

```
;;---------------------------------------------------------------------------------------
;; POINT FUNCTIONS
;;---------------------------------------------------------------------------------------
;;---------------------------------------------------------------------------------------

;conversion routines
(defun deg-to-rad (degree) (* degree 0.0174532925199432946))
(defun rad-to-deg (radian) (* radian 57.2957795130823229))

;access routines
(defun ptx (pt) (nth 0 pt))
(defun pty (pt) (nth 1 pt))
(defun ptz (pt) (nth 2 pt))

;routine to test if two points are equal with form (x y z)
;(equal-points '(1 1 1) '(2 2 2))
;(equal-points '#(1 1 1) '#(2 2 2))
(defun equal-points (pt1 pt2)
  (setq pt1 (coerce pt1 'list))
  (setq pt2 (coerce pt2 'list))
  (and (eql (ptx pt2) (ptx pt1)) (eql (pty pt2) (pty pt1)) (eql (ptz pt2) (ptz pt1))))

;routine to add two points with form (x y z)
;(add-points '(1 1 1) '(2 2 2))
;(add-points '#(1 1 1) '#(2 2 2))
(defun add-points (pt1 pt2)
  (setq pt1 (coerce pt1 'list))
  (setq pt2 (coerce pt2 'list))
  (list (+ (ptx pt1) (ptx pt2)) (+ (pty pt1) (pty pt2)) (+ (ptz pt1) (ptz pt2))))

;routine to minus two points with form (x y z)
;(minus-points '(1 1 1) '(2 2 2))
;(minus-points '#(1 1 1) '#(2 2 2))
(defun minus-points (pt1 pt2)
  (setq pt1 (coerce pt1 'list))
  (setq pt2 (coerce pt2 'list))
  (list (- (ptx pt1) (ptx pt2)) (- (pty pt1) (pty pt2)) (- (ptz pt1) (ptz pt2))))

;routine to get the distance between two points with form (x y z)
;(distance-between-points '(1 1 1) '(2 2 2))
;(distance-between-points '#(1 1 1) '#(2 2 2))
(defun distance-between-points (pt1 pt2)
  (setq pt1 (coerce pt1 'list))
  (setq pt2 (coerce pt2 'list))
  (sqrt (+ (expt (- (ptx pt2) (ptx pt1)) 2)
           (expt (- (pty pt2) (pty pt1)) 2) (expt (- (ptz pt2) (ptz pt1)) 2))))

;routine to get the angle between two points with form (x y z) (returns degrees)
;(angle-between-points '(1 1 1) '(2 2 2))
;(angle-between-points '#(1 1 1) '#(2 2 2))
(defun angle-between-points (pt1 pt2 &aux base)
  (if (not (equal-points pt1 pt2))
      (progn
        (setq pt1 (coerce pt1 'list))
        (setq pt2 (coerce pt2 'list))
;make base value
        (setq base
            (rad-to-deg (acos (/ (- (ptx pt2) (ptx pt1)) (distance-between-points pt1
pt2)))))
        (if (<= (pty pt1) (pty pt2))
            base
```

```
                (- 0.0 base)))
        0.0))

;compute-polygon-area
;Takes a list of 3D points (z values are ignored).
;For example, a polygon with 3 vertices the area:
;A = 1/2(y1 + y2)(x2 - x1) +
;    1/2(y2 + y3)(x3 - x2) +
;    1/2(y3 + y1)(x1 - x3)
;NOTE: Assumes that the list contains points in order. Error checking to ensure
;this is not done for efficiency sakes.
(defun compute-polygon-area (pts-list)
  (let ((result 0)
      (curpos 0)
      (len (length pts-list)))
    (loop
      (when (>= curpos len) (return (abs result)))
      (let ((pt1 (nth curpos pts-list))
            (pt2 (if (= curpos (1- len)) (car pts-list) (nth (1+ curpos) pts-list))))
        (setf result (+ result (* 0.5 (+ (pty pt1) (pty pt2)) (- (ptx pt2) (ptx pt1)))))
        (incf curpos)))))

;places a point vector into a 4x1 matrix for use in mult-41matrix
(defun make-point-matrix (point)
  (let ((temp (make-array '(4 1))))
        (setf (aref temp 0 0) (nth 0 (coerce point 'list)))
        (setf (aref temp 1 0) (nth 1 (coerce point 'list)))
        (setf (aref temp 2 0) (nth 2 (coerce point 'list)))
        (setf (aref temp 3 0) 1)
        temp))

;;-------------------------------------------------------------------------------
;;-------------------------------------------------------------------------------
;; MATRIX FUNCTIONS
;;-------------------------------------------------------------------------------
;;-------------------------------------------------------------------------------

(defun mult-33matrix (mat1 mat2 &aux mat3 temp)
  (setf mat3 (make-array '(3 3)))
  (do ((i 0 (+ i 1))) ((= i 3))
        (do ((j 0 (+ j 1))) ((= j 3))
          (setf temp 0)
          (do ((k 0 (+ k 1))) ((= k 3))
               (setf temp (+ temp (* (aref mat1 i k) (aref mat2 k j)))))
          (setf (aref mat3 i j) temp)))
  mat3)

;multiplies two 4x4 matrices and returns the resultant 4x4
(defun mult-44matrix (mat1 mat2 &aux mat3 temp)
  (setf mat3 (make-array '(4 4)))
  (do ((i 0 (+ i 1))) ((= i 4))
        (do ((j 0 (+ j 1))) ((= j 4))
          (setf temp 0)
          (do ((k 0 (+ k 1))) ((= k 4))
               (setf temp (+ temp (* (aref mat1 i k) (aref mat2 k j)))))
          (setf (aref mat3 i j) temp)))
  mat3)

;mat1 is 4x4, mat2 is 4x1, result is 4x1
(defun mult-41matrix (mat1 mat2 &aux mat3 temp)
  (setf mat3 (make-array '(4 1)))
```

```
  (do ((i 0 (+ i 1))) ((= i 4))
        (do ((j 0 (+ j 1))) ((= j 1))
          (setf temp 0)
          (do ((k 0 (+ k 1))) ((= k 4))
               (setf temp (+ temp (* (aref mat1 i k) (aref mat2 k j)))))
          (setf (aref mat3 i j) temp)))
  mat3)

(defun ludcmp (a n indx d &aux vv big temp sum dum imax)
  (setf d 1.0)
  (setf vv (make-array n))
  (setf indx (make-array n))

;get largest value in a row
  (do ((i 0 (+ i 1))) ((= i n))
        (setf big 0.0)
        (do ((j 0 (+ j 1))) ((= j n))
          (if (> (setf temp (abs (aref a i j))) big) (setf big temp)))
        (if (= big 0.0) (print "error: singular matrix in ludcmp"))
        (setf (aref vv i) (/ 1.0 big)))

;loop through columns
  (do ((j 0 (+ j 1))) ((= j n))

;go down rows until diagonal element
        (do ((i 0 (+ i 1))) ((not (< i j)))
          (setf sum (aref a i j))
          (do ((k 0 (+ k 1))) ((not (< k i)))
               (setf sum (- sum (* (aref a i k) (aref a k j)))))
          (setf (aref a i j) sum))

        (setf big 0.0)
;start at the diagonal element
        (do ((i j (+ i 1))) ((= i n))
          (setf sum (aref a i j))
          (do ((k 0 (+ k 1))) ((not (< k j)))
               (setf sum (- sum (* (aref a i k) (aref a k j)))))
          (setf (aref a i j) sum)
          (if (>= (setf dum (* (aref vv i) (abs sum))) big)
               (progn (setf big dum) (setf imax i))))

;interchange rows
        (if (/= j imax)
          (progn
               (do ((k 0 (+ k 1))) ((= k n))
                 (setf dum (aref a imax k))
                 (setf (aref a imax k) (aref a j k))
                 (setf (aref a j k) dum))
               (setf d (- 0 d))
               (setf dum (aref vv imax))
               (setf (aref vv imax) (aref vv j))
               (setf (aref vv j) dum)))

        (setf (aref indx j) imax)
        (if (= (aref a j j) 0.0) (setf (aref a j j) 0.00000000000000000000001))
        (if (/= j (- n 1))
          (progn
               (setf dum (/ 1.0 (aref a j j)))
               (do ((i (+ j 1) (+ i 1))) ((= i n))
                 (setf (aref a i j) (* (aref a i j) dum)))))
```

```
  )
(list a indx d)
)

(defun lubksb (a n indx b &aux ii ip sum)
  (setf ii -1)
  (do ((i 0 (+ i 1))) ((= i n))
        (setf ip (aref indx i))
        (setf sum (aref b ip))
        (setf (aref b ip) (aref b i))
        (if (/= ii -1)
          (do ((j ii (+ j 1))) ((not (<= j ( - i 1))))
               (setf sum (- sum (* (aref a i j) (aref b j)))))
          (if (/= sum 0.0) (setf ii i)))
        (setf (aref b i) sum))

  (do ((i (- n 1) (- i 1))) ((< i 0))
        (setf sum (aref b i))
        (do ((j (+ i 1) (+ j 1))) ((= j n))
          (setf sum (- sum (* (aref a i j) (aref b j)))))
        (setf (aref b i) (/ sum (aref a i I))))
  b
)

;get inverse of given matrix
(defun matinv (a n &aux result col y indx d)
  (setf result (ludcmp a n nil nil))
  (setf col (make-array n))
  (setf y (make-array `(,n ,n)))
  (do ((j 0 (+ j 1))) ((= j n))
        (do ((i 0 (+ i 1))) ((= i n)) (setf (aref col i) 0.0))
        (setf (aref col j) 1.0)
        (setf col (lubksb a n (nth 1 result) col))
        (do ((i 0 (+ i 1))) ((= i n)) (setf (aref y i j) (aref col i))))
  y
)

;;-----------------------------------------------------------------------------
;;-----------------------------------------------------------------------------
;; GENERIC HANDLERS
;;-----------------------------------------------------------------------------
;;-----------------------------------------------------------------------------

;*****************************************************************************
(define-handler (CAD-SHAPE-OBJECT get-reference-inst) (&aux items)
  (if (setq items (cad-linked-items (to-items self))) (car items)))

;*****************************************************************************
;places instance origin and rotation into a 4x4 matrix
(define-handler (CAD-SHAPE-OBJECT build-matrix) ()
  (let ((temp (make-array '(4 4)))
        angle)
        (setf (aref temp 0 0) 1)
        (setf (aref temp 1 1) 1)
        (setf (aref temp 2 2) 1)
        (setf (aref temp 3 3) 1)
;put in origin
        (setf (aref temp 0 3) (nth 0 (coerce (slot-value self 'origin) 'list)))
        (setf (aref temp 1 3) (nth 1 (coerce (slot-value self 'origin) 'list)))
        (setf (aref temp 2 3) (nth 2 (coerce (slot-value self 'origin) 'list)))
;put in z rotation
```

```
        (setf angle (deg-to-rad (nth 2 (coerce (slot-value self 'rotation) 'list))))
        (setf (aref temp 0 0) (cos angle))
        (setf (aref temp 1 1) (cos angle))
        (setf (aref temp 0 1) (- 0.0 (sin angle)))
        (setf (aref temp 1 0)        (sin angle))
;return temp
        temp))

;*****************************************************************************************
;gets the transformation matrix of an instance - traces down through linked items
(define-handler (CAD-SHAPE-OBJECT get-matrix) ()
  (if (setq temp (send-msg self :get-reference-inst))
        (mult-44matrix (send-msg temp :get-matrix) (send-msg self :build-matrix))
        (send-msg self :build-matrix)))

;*****************************************************************************************
(define-handler (CAD-SHAPE-OBJECT get-relative-origin) ()
  (slot-value self 'origin))

;*****************************************************************************************
;get an objects absolute origin - if there is a reference object, then get it's
transformation
;matrix and multiply our origin by that
(define-handler (CAD-SHAPE-OBJECT get-absolute-origin) (&aux temp)
  (if (setq temp (send-msg self :get-reference-inst))
    (let ((result (mult-41matrix
                    (send-msg temp :get-matrix) (make-point-matrix (slot-value self
'origin)))))
      (list (aref result 0 0) (aref result 1 0) (aref result 2 0)))
    (send-msg self :get-relative-origin)))

;*****************************************************************************************
(define-handler (CAD-SHAPE-OBJECT set-relative-origin) (new)
  (setf (slot-value self 'origin) new))

;*****************************************************************************************
;set an objects absolute origin - if there is a reference object, then get it's
transformation
;matrix and multiply the new origin by the inverse of that
(define-handler (CAD-SHAPE-OBJECT set-absolute-origin) (new &aux temp)
  (if (setq temp (send-msg self :get-reference-inst))
        (let ((result (mult-41matrix
                (matinv (send-msg temp :get-matrix) 4) (make-point-matrix new))))
          (send-msg self :set-relative-origin
                (list (aref result 0 0) (aref result 1 0) (aref result 2 0))))
        (send-msg self :set-relative-origin new)))

;*****************************************************************************************
(define-handler (CAD-SHAPE-OBJECT get-relative-rotation) ()
  (slot-value self 'rotation))

;*****************************************************************************************
(define-handler (CAD-SHAPE-OBJECT get-absolute-rotation) (&aux temp)
  (if (setq temp (send-msg self :get-reference-inst))
        (add-points (send-msg self :get-relative-rotation) (send-msg temp
        :get-absolute-rotation))
        (send-msg self :get-relative-rotation)))

;*****************************************************************************************
(define-handler (CAD-SHAPE-OBJECT set-relative-rotation) (new)
  (setf (slot-value self 'rotation) new))
```

```
;*****************************************************************************************
(define-handler (CAD-SHAPE-OBJECT set-absolute-rotation) (new &aux temp)
  (if (setq temp (send-msg self :get-reference-inst))
        (send-msg self :set-relative-rotation
              (minus-points new (send-msg temp :get-absolute-rotation)))
        (send-msg self :set-relative-rotation new)))

;;-----------------------------------------------------------------------------------------
;;-----------------------------------------------------------------------------------------
;; AUTOCAD SPECIFIC HANDLERS
;;-----------------------------------------------------------------------------------------
;;-----------------------------------------------------------------------------------------

;*****************************************************************************************
;Handler responsible for drawing object, setting entity handle, and linking to object in
CAD.
;Return t if successful, else ().
;NOTE: ASSUMING THAT ORIGIN HAS BEEN SET!
(define-handler (CAD-SHAPE-OBJECT acad-draw) ()
  (if (slot-value self 'origin)
        (progn
          (send-msg self :acad-cleanup)
          (send-msg self :acad-build `(insert-text (quote ,(send-msg self
          :get-absolute-origin))
               ,(symbol-name (gw-name self))))
          t)))

;*****************************************************************************************
(define-handler (CAD-SHAPE-OBJECT acad-object-moved) (offset)
  (print (format nil "~S moved in AutoCad ~S." (gw-name self) offset))
;update origin slot - when modified called
  (send-msg self :set-absolute-origin (add-points (send-msg self :get-absolute-origin)
offset))
  t)

;;-----------------------------------------------------------------------------------------
;;-----------------------------------------------------------------------------------------
;; MICROSTATION SPECIFIC HANDLERS
;;-----------------------------------------------------------------------------------------
;;-----------------------------------------------------------------------------------------

;*****************************************************************************************
;Handler responsible for drawing object, setting entity handle, and linking to object in
CAD.
;Return t if successful, else ().
(define-handler (CAD-SHAPE-OBJECT mstation-draw) (&aux tag)
  (if (slot-value self 'origin)
        (progn
          (send-msg self :mstation-cleanup)
          (send-msg self :mstation-build (format NIL "insert_text ~A ~S ~S"
               (make-list-string (send-msg self :get-absolute-origin)) (gw-name self)
(gw-name self)))
          t)))

;;-----------------------------------------------------------------------------------------
;;-----------------------------------------------------------------------------------------
;; DEFINE LINE-CAD-SHAPE-OBJECT
;;-----------------------------------------------------------------------------------------
;;-----------------------------------------------------------------------------------------

(DEFINE-FRAME LINE-CAD-SHAPE-OBJECT
```

```
   (:is CAD-SHAPE-OBJECT)
   (END
        :doc-string "End of line relative to origin."
        :when-modified (cad-update-linked))
)

;*****************************************************************************************
;Handler responsible for drawing object, setting entity handle, and linking to object in
CAD.
;Return t if successful, else ().
;NOTE: ASSUMING THAT ORIGIN, ROTATION, AND END HAS BEEN SET!
(define-handler (LINE-CAD-SHAPE-OBJECT acad-draw) (&aux true-origin)
  (if (and (slot-value self 'origin) (slot-value self 'rotation) (slot-value self 'end))
        (progn
          (send-msg self :acad-cleanup)
          (setf true-origin (send-msg self :get-absolute-origin))
          (send-msg self :acad-build (append `(insert-line (quote ,true-origin)
                ,(add-points true-origin (slot-value self 'end)))))
          t)))

;;-----------------------------------------------------------------------------------------
;;-----------------------------------------------------------------------------------------
;; DEFINE 2D-POLYGON-CAD-SHAPE-OBJECT
;;-----------------------------------------------------------------------------------------
;;-----------------------------------------------------------------------------------------

(DEFINE-FRAME 2D-POLYGON-CAD-SHAPE-OBJECT
   (:is CAD-SHAPE-OBJECT)
   (CORNERS
        :doc-string "Corners of polygon relative to origin."
        :when-modified (cad-update))
)

;*****************************************************************************************
;Handler responsible for drawing object, setting entity handle, and linking to object in
CAD.
;Return t if successful, else ().
;NOTE: ASSUMING THAT ORIGIN, ROTATION, AND CORNERS HAVE BEEN SET!
(define-handler (2D-POLYGON-CAD-SHAPE-OBJECT acad-draw) (&aux true-origin mod-corners)
   (if (and (slot-value self 'origin) (slot-value self 'rotation) (slot-value self
'corners))
        (progn
          (send-msg self :acad-cleanup)
          (setf true-origin (send-msg self :get-absolute-origin))
;convert relative corners to absolute
          (setf mod-corners
                (mapcar #'(lambda (pt) (add-points true-origin pt)) (slot-value self
'corners)))
          (send-msg self :acad-build (append `(insert-2d-polygon (quote ,mod-corners))))
          t)))

;;-----------------------------------------------------------------------------------------
;;-----------------------------------------------------------------------------------------
;; DEFINE 3D-POLYGON-CAD-SHAPE-OBJECT
;;-----------------------------------------------------------------------------------------
;;-----------------------------------------------------------------------------------------

(DEFINE-FRAME 3D-POLYGON-CAD-SHAPE-OBJECT
   (:is CAD-SHAPE-OBJECT)
   (HEIGHT
        :when-modified (cad-update))
```

```
   (CORNERS
        :doc-string "Corners of polygon relative to origin."
        :when-modified (cad-update))
)

;*****************************************************************************************
;set the corners slot
(define-handler  (3D-POLYGON-CAD-SHAPE-OBJECT  set-absolute-corners)    (ptList  &aux
true-origin)
  (setf true-origin (send-msg self :get-absolute-origin))
  (setf (slot-value self 'corners) (mapcar #'(lambda (pt) (minus-points pt true-origin))
ptList)))

;*****************************************************************************************
;Handler responsible for drawing object, setting entity handle, and linking to object in
CAD.
;Return t if successful, else ().
;NOTE: ASSUMING THAT ORIGIN, ROTATION, CORNERS, AND HEIGHT HAVE BEEN SET!
(define-handler (3D-POLYGON-CAD-SHAPE-OBJECT acad-draw) (&aux true-origin mod-corners)
  (if (and (slot-value self 'origin) (slot-value self 'rotation)
        (slot-value self 'corners) (slot-value self 'height))
        (progn
          (send-msg self :acad-cleanup)
          (setf true-origin (send-msg self :get-absolute-origin))
;convert relative corners to absolute
          (setf mod-corners
                (mapcar #'(lambda (pt) (add-points true-origin pt)) (slot-value self
'corners)))
          (send-msg self :acad-build (append `(insert-3d-polygon (quote ,mod-corners)
               ,(slot-value self 'height))))
          t)))

(define-handler (3D-POLYGON-CAD-SHAPE-OBJECT mstation-draw) (&aux true-origin mod-corners
tag)
  (if (and (slot-value self 'origin) (slot-value self 'rotation)
        (slot-value self 'corners) (slot-value self 'height))
        (progn
          (send-msg self :mstation-cleanup)
          (setf true-origin (send-msg self :get-absolute-origin))
;convert relative corners to absolute
          (setf mod-corners
                (mapcar #'(lambda (pt) (add-points true-origin pt)) (slot-value self
'corners)))
          (send-msg self :mstation-build (format NIL "insert_3d_polygon ~A ~A ~S ~S"
               (length mod-corners)
               (list-make-list-string mod-corners)
               (slot-value self 'height)
               (gw-name self)))
          t)))

;;-----------------------------------------------------------------------------------------
;;-----------------------------------------------------------------------------------------
;; DEFINE 3D-CUBE-CAD-SHAPE-OBJECT
;;-----------------------------------------------------------------------------------------
;;-----------------------------------------------------------------------------------------

(DEFINE-FRAME 3D-CUBE-CAD-SHAPE-OBJECT
   (:is CAD-SHAPE-OBJECT)
   (X_LENGTH
        :doc-string "X length relative to origin."
        :when-modified (cad-update))
```

```
   (Y_LENGTH
        :doc-string "Y length relative to origin."
        :when-modified (cad-update))
   (Z_LENGTH
        :doc-string "Z length relative to origin."
        :when-modified (cad-update))
)

;*****************************************************************************************
;Handler responsible for drawing object, setting entity handle, and linking to object in
CAD.
;Return t if successful, else ().
;NOTE: ASSUMING THAT ORIGIN, ROTATION, X_LENGTH, Y_LENGTH, AND Z_LENGTH HAS BEEN SET!
(define-handler (3D-CUBE-CAD-SHAPE-OBJECT acad-draw) ()
  (if (and (slot-value self 'origin) (slot-value self 'rotation)
           (slot-value self 'x_length) (slot-value self 'y_length) (slot-value self
'z_length))
        (progn
          (send-msg self :acad-cleanup)
          (send-msg self :acad-build (append `(insert-3dcube-main
               (quote ,(send-msg self :get-absolute-origin))
               ,(slot-value self 'x_length)
               ,(slot-value self 'y_length)
               ,(slot-value self 'z_length)
               ,(nth 2 (send-msg self :get-absolute-rotation)))))
          t)))

(define-handler (3D-CUBE-CAD-SHAPE-OBJECT mstation-draw) (&aux tag)
  (if (and (slot-value self 'origin) (slot-value self 'rotation)
           (slot-value self 'x_length) (slot-value self 'y_length) (slot-value self
'z_length))
        (progn
          (send-msg self :mstation-cleanup)
          (send-msg self :cad-setup)
          (send-msg self :mstation-build (format NIL "insert_3d_cube ~A ~S ~S ~S ~S"
               (make-list-string (send-msg self :get-absolute-origin))
               (slot-value self 'x_length)
               (slot-value self 'y_length)
               (slot-value self 'z_length)
               (nth 2 (send-msg self :get-absolute-rotation))
               (gw-name self)))
          t)))

;;-----------------------------------------------------------------------------------------
;;-----------------------------------------------------------------------------------------
;; DEFINE 3DFACE-CAD-SHAPE-OBJECT
;;-----------------------------------------------------------------------------------------
;;-----------------------------------------------------------------------------------------

(DEFINE-FRAME 3DFACE-CAD-SHAPE-OBJECT
   (:is CAD-SHAPE-OBJECT)
   (HEIGHT
        :when-modified (cad-update))
   (END
        :doc-string "End of line relative to origin."
        :when-modified (cad-update-linked))
)

;*****************************************************************************************
;Handler responsible for drawing object, setting entity handle, and linking to object in
CAD.
```

```
;Return t if successful, else ().
;NOTE: ASSUMING THAT ORIGIN, ROTATION, AND END HAS BEEN SET!
(define-handler (3DFACE-CAD-SHAPE-OBJECT acad-draw) (&aux true-origin)
  (if (and (slot-value self 'origin) (slot-value self 'rotation)
           (slot-value self 'end) (slot-value self 'height))
           (progn
             (send-msg self :acad-cleanup)
             (setf true-origin (send-msg self :get-absolute-origin))
             (send-msg self :acad-build (append `(insert-3dface
                  (quote ,true-origin) ,(add-points true-origin (slot-value self 'end))
                  ,(slot-value self 'height))))
             t)))
```

## CAD Sytem Source Code

### *AutoCAD*

File: acadctrl.lsp

Contains basic extended entity data functions and object display functions.

```
;;*************************************************************************
;;-------------------------------------------------------------------------
;; Extended Entity Data Routines
;;-------------------------------------------------------------------------
;;*************************************************************************
(defun c:all-xdata () (entget (car (entsel)) (list "*")))

;;*************************************************************************
;Select and print extended entity data of an entity.
(defun c:ct-data ( / entity xdata)
  (if (setq entity (car (entsel "Select object: ")))
    (if (setq xdata (get-xdata entity))
      (progn (princ "\nExtended entity data for CADtalk is: ") (print xdata))
      (princ "\nNo extended entity data for CADtalk\n"))
    (princ "\nNo Object Selected."))
  *nothing*)

;;*************************************************************************
;Add extended entity data to given entity, inst_name is text string of data.
(defun add-xdata (ename inst_name)
  (plist-put ename "INST-NAME" inst_name)
  t)

;;*************************************************************************
;Get extended entity data of given entity.
(defun get-xdata (ename)
  (plist-get ename "INST-NAME"))

;;*************************************************************************
;;-------------------------------------------------------------------------
;; Utility routines called by GW to draw an object
;;
;; NOTE: Routines MUST return entity handle within a list!
;;-------------------------------------------------------------------------
;;*************************************************************************
;(defun *error* (msg) *nothing*)
```

```
;(setq *error* nil)

;;*****************************************************************************
;returns list of entity handle - requires some user interaction
;(insert-text '(5 5 0) "hi")
(defun insert-text (pt1 text / cmd_var)
  (if (null pt1) (setq pt1 pause))
  (setq cmd_var (getvar "CMDECHO"))
  (setvar "CMDECHO" 1)
  (command "_.TEXT" pt1 pause pause text)
  (setvar "CMDECHO" cmd_var)
;Return values.
  (list (hanlast)))

;;*****************************************************************************
;returns list with: (<handle list> <start pt list> <end pt list>)
;(insert-line '(0 0 0) '(10 10 0))
(defun insert-line (pt1 pt2 / cmd_var)
  (if (null pt1) (setq pt1 (getpoint "From Point:")))
  (if (null pt2) (setq pt2 (getpoint pt1 "\nTo Point:")))
  (setq cmd_var (getvar "CMDECHO"))
  (setvar "CMDECHO" 1)
  (command "_.LINE" pt1 pt2 "")
  (setvar "CMDECHO" cmd_var)
;Return values.
  (list (list (hanlast)) pt1 pt2))

;;*****************************************************************************
;returns entity handle of polyline
;(insert-2d-polygon '((13000 -1800 0) (18000 -5900 0) (13000 -8000 0)))
(defun insert-2d-polygon (corners / cmd_var)
  (setq cmd_var (getvar "CMDECHO"))
  (setvar "CMDECHO" 0)
  (command "_.PLINE")
  (foreach pt corners (command pt))
  (command "cl")
  (setvar "CMDECHO" cmd_var)
;Return values.
  (list (hanlast)))

;;*****************************************************************************
;returns entity handle of polyline
;(insert-3d-polygon '((13000 -1800 0) (18000 -5900 0) (13000 -8000 0)) '500)
(defun insert-3d-polygon (corners height / i numCorners comm cmd_var)

  (defun add-height (corners height)
    (if (null corners)
      nil
      (cons (list (caar corners) (cadar corners) (+ (caddar corners) height))
      (add-height (cdr corners) height))))

  (if (null height) (setq height (getint "Enter height: ")))

  (setq numCorners (length corners))
;setup points
  (setq comm (append '(command) '("_.PFACE") corners (add-height corners height) '("")))
;add lower face
  (setq i 1)
  (repeat numCorners
    (setq comm (append comm (list i)))
    (setq i (+ i 1)))
```

```
  (setq comm (append comm '("")))
;add upper face
  (setq i (+ 1 numCorners))
  (repeat numCorners
    (setq comm (append comm (list i)))
    (setq i (+ i 1)))
  (setq comm (append comm '("")))
;add side faces
  (setq i 1)
  (repeat numCorners
    (if (= i numCorners)
      (setq comm (append comm (list i 1 (+ 1 numCorners) (+ i numCorners)) '("")))
       (setq comm (append comm (list i (+ i 1) (+ i 1 numCorners) (+ i numCorners))
'(""))))
    (setq i (+ i 1)))
;add final return
  (setq comm (append comm '("")))
;quote points
  (setq comm (mapcar '(lambda (x) (if (listp x) (cons 'quote (list x)) x)) comm))
;evaluate command
  (setq cmd_var (getvar "CMDECHO"))
  (setvar "CMDECHO" 0)
  (eval comm)
  (setvar "CMDECHO" cmd_var)
;Return values.
  (list (hanlast))
)

;;*****************************************************************************
;pt1 is origin, l is length(x), w is width(y), h is height(z), a is rotation angle(z
rotation in AUNITS mode)
;pt1 is smallest x, median y
;ex. of draw:
;   -------------------------------
;   |                              |
;   |                              |
;   * < = pt1                      |
;   |                              |
;   |                              |
;   -------------------------------
(defun insert-3dcube-main (pt1 l w h a / cmd_var origin pt2 pt3 pt4)
  (setq origin pt1) ;save origin for rotation
  (setq pt1 (list (car pt1) (- (cadr pt1) (/ w 2.0)) (caddr pt1)))
  (setq pt2 (list (car pt1) (+ (cadr pt1) w) (caddr pt1)))
  (setq pt3 (list (+ (car pt1) l) (+ (cadr pt1) w) (caddr pt1)))
  (setq pt4 (list (+ (car pt1) l) (cadr pt1) (caddr pt1)))
;Draw cube.
  (insert-3d-polygon (list pt1 pt2 pt3 pt4) h)
;Rotate into position.
  (setq cmd_var (getvar "CMDECHO"))
  (setvar "CMDECHO" 0)
  (command "_.ROTATE" (entlast) "" origin a)
  (setvar "CMDECHO" cmd_var)
;Return values.
  (list (hanlast))
)

;;*****************************************************************************
;returns list with: (<handle list> <start pt list of centerline> <end pt list of
centerline>)
;(insert-3dcube '2 '20)
```

```
(defun insert-3dcube (width height / pt1 pt2)
;get centerline for cube
  (setq pt1 (getpoint "From point: "))
  (setq pt2 (getpoint pt1 "\nTo point: "))
  (if (null width) (setq width (getint "Enter width: ")))
  (if (null height) (setq height (getint "Enter height: ")))
  (insert-3dcube-main pt1 (distance pt1 pt2) width height
     (read (angtos (angle pt1 pt2) (getvar "AUNITS") 5)))
;Return values.
  (list (list (hanlast)) pt1 pt2))

;;*************************************************************************
;returns list with: (<handle list> <start pt list> <end pt list>)
(defun insert-3dface-main (pt1 pt2 pt3 pt4)
  (command "_.3DFACE" pt1 pt2 pt3 pt4 "")
;Return values.
  (list (list (hanlast)) pt1 pt2))

;;*************************************************************************
;pt1 = origin, pt2 = end
;returns list with: (<handle list> <start pt list> <end pt list>)
(defun insert-3dface (pt1 pt2 height / pt3 pt4)
  (if (null pt1) (setq pt1 (getpoint "From point: ")))
  (if (null pt2) (setq pt2 (getpoint pt1 "\nTo point: ")))
  (if (null height) (setq height (getint "Enter height: ")))
  (setq pt3 (list (car pt2) (cadr pt2) (+ height (caddr pt2))))
  (setq pt4 (list (car pt1) (cadr pt1) (+ height (caddr pt1))))
  (insert-3dface-main pt1 pt2 pt3 pt4))
```

### *Microstation*

File: cadcmds.mc

Contains the Microstation Development Language (MDL) C code for Microstation commands. This file is compiled and linked into an MDL application file that is loaded into the Microstation system. Note that results are returned to ACE through MDL system variable CBRAIN RESULTS.

```
/*-------------------------------------------------------------------+
|                                                                    |
|   CADCMDS.MC  -- Microstation CAD Drawing Routines.                |
|                                                                    |
+-------------------------------------------------------------------*/

/*-------------------------------------------------------------------+
|   Include Files                                                    |
+-------------------------------------------------------------------*/
#include <mdl.h>
#include <stdarg.h>
#include <dlogbox.h>
#include <dlogitem.h>
#include <mdlio.h>
#include <rscdefs.h>
#include <msinputq.h>

#include <tcb.h>
```

```
#include <global.h>
#include <mselems.h>
#include <userfnc.h>
#include <userpref.h>
#include <widechar.h>
#include <mselemen.fdf>
#include <cexpr.h>

#include "cadcmds.h"
#include "cmd.h"
#include "dialog.h"

/*----------------------------------------------------------------------+
|   Private Global variables                                            |
+----------------------------------------------------------------------*/
#define CADCMDSTAG  98 /* Unique ID to identify our extended data. */

extern DialogHookInfo uHooks[];

int numShapePts;
Dpoint3d shapePts[10];
int listBoxItemSelected;
ULong curFilePos, curTag;
MSElementUnion curElement;
char instName[50], resultBuf[250];
double curWidth, curLength, curHeight;
double dialogx1, dialogx2, dialogx3, dialogx4;
double dialogy1, dialogy2, dialogy3, dialogy4;
double dialogz1, dialogz2, dialogz3, dialogz4;

/*----------------------------------------------------------------------+
|   Routine Definitions                                                 |
+----------------------------------------------------------------------*/
/*----------------------------------------------------------------------+
+----------------------------------------------------------------------*/
double getAngle(Dpoint3d *pt1, Dpoint3d *pt2) {
      if (pt1->y <= pt2->y)
               return acos( (pt2->x - pt1->x) / mdlVec_distance(pt1, pt2) );
      else {
               if (pt2->x < pt1->x)
                        return acos( (pt2->x - pt1->x) / mdlVec_distance(pt1, pt2) ) +
fc_piover2;
               else if (pt2->x == pt1->x)
                        return acos( (pt2->x - pt1->x) / mdlVec_distance(pt1, pt2) ) +
fc_pi;
               else
                        return acos( (pt2->x - pt1->x) / mdlVec_distance(pt1, pt2) ) -
fc_piover2;
      }
}

/*----------------------------------------------------------------------+
+----------------------------------------------------------------------*/
void addTag(ULong filePos) {
char temp[10];
      if (mdlAssoc_tagElement(&curTag, filePos, 0) != SUCCESS)
               printf("error tagging element\n");
      else {
               printf("element tagged with %ld\n", curTag);
               sprintf(temp, "%ld", curTag);
               mdlSystem_putenv("TAG_VALUE", temp);
```

```
    }
}

/*-------------------------------------------------------------------+
+-------------------------------------------------------------------*/
void getTag(MSElementUnion *element) {
char temp[10];
    if (mdlAssoc_isTagged(&curTag, element)) {
            printf("element tagged with %ld\n", curTag);
            sprintf(temp, "%ld", curTag);
            mdlSystem_putenv("TAG_VALUE", temp);
    }
    else
            printf("element not tagged\n");
}

/*-------------------------------------------------------------------+
+-------------------------------------------------------------------*/
void addXDataName(MSElement *element, char *name) {
gwData attrData;
LinkageHeader linkHdr;

    memset(&attrData, '\0', sizeof(gwData));
    memset(&linkHdr, '\0', sizeof(LinkageHeader));
    linkHdr.primaryID = CADCMDSTAG;
    linkHdr.user = 1;
    strcpy(attrData.instName, name);
    if (getXDataName(element))
            mdlLinkage_deleteFromElement(element, CADCMDSTAG, 1, NULL, NULL, NULL);
    mdlLinkage_appendToElement(element, &linkHdr, &attrData, 1, NULL);
}

/*-------------------------------------------------------------------+
+-------------------------------------------------------------------*/
int getXDataName(MSElement *element) {
typedef struct attrUnion {
    LinkageHeader linkHdr;
    gwData attrData;
    } attrInfo;
attrInfo attribs;

    if (mdlLinkage_extractFromElement(&attribs, element, CADCMDSTAG, 1, NULL, NULL,
NULL) ==
             NULL) {
            printf("no linkage info found\n");
            return FALSE; }
    strcpy(instName, attribs.attrData.instName);
    return TRUE;
}

/*-------------------------------------------------------------------+
| name          elmDscr_show - useful function to dump an elm descr  |
+-------------------------------------------------------------------*/
Public int elmDscr_show(MSElementDescr *elmDscrP, char *currentIndent) {
char indent[128];
int color, weight, style, level, ggNum, class, locked, new,
    modified, viewIndepend, solidHole;

    if (elmDscrP == NULL) return SUCCESS;

    strcpy(indent, currentIndent);
```

```
        strcat(indent, "|   ");

        do {
                mdlElement_getSymbology(&color, &weight, &style, &elmDscrP->el);
                mdlElement_getProperties(&level, &ggNum, &class, &locked, &new,
                        &modified, &viewIndepend, &solidHole, &elmDscrP->el);
                printf("%shdr=%d,typ=%d,cmplx=%d, ", currentIndent,
                        elmDscrP->h.isHeader, elmDscrP->el.hdr.ehdr.type,
                        elmDscrP->el.hdr.ehdr.complex);
                printf("c=%d,w=%d,s=%d,l=%d,gg=%d,cl=%d\n",
                        color, weight, style, level, ggNum, class);

                if (elmDscrP->h.isHeader) {
                        elmDscr_show(elmDscrP->h.firstElem, indent);
                        printf("%send of chain\n", currentIndent);
                }

                elmDscrP = elmDscrP->h.next;
        } while (elmDscrP);
}

/*----------------------------------------------------------------------+
| name          dataButtonHit                                           |
+----------------------------------------------------------------------*/
Private void dataButtonHit(Dpoint3d *point, int view) {
ULong tag;
int segment;
Dpoint3d closestPoint;

        curFilePos = mdlLocate_findElement(point, view, 0, 0, FALSE);
        if (!curFilePos)
                mdlOutput_printf(MSG_STATUS, "Element not found");
        else {
                mdlOutput_printf(MSG_STATUS, "ELEMENT FOUND: type=%d",
                        dgnBuf->ehdr.type);

/* MODIFIED TO ALSO DISPLAY XDATA NAME AND TAG */
                getXDataName((MSElement *)dgnBuf);
                getTag((MSElementUnion *)dgnBuf);

                if (dgnBuf->ehdr.type == LINE_STRING_ELM) {
                        mdlLocate_getProjectedPoint(&closestPoint, &segment, NULL);
                        mdlOutput_printf(MSG_PROMPT,
                                "Locate point=[%d,%d,%d], segment=%d",
                                closestPoint.x, closestPoint.y, closestPoint.z, segment);
                }
        }
}

/*----------------------------------------------------------------------+
| name      locateElement - print out element type of elements pointed  |
|             to by user.                                               |
+----------------------------------------------------------------------*/
Public cmdName locateElement(void) cmdNumber CMD_PICKIT {
        mdlState_startPrimitive(dataButtonHit, locateElement, 0, 0);

        /* reset the location logic */
        mdlLocate_init();

        /* allow any element */
        mdlLocate_allowLocked();
```

```
      /* change the cursor to be the "locate" cursor */
      mdlLocate_setCursor();
}

/*----------------------------------------------------------------------+
+   PARAMETERS: center = center point of cube
+   NOTE:
+       This routine assumes that the following variables are set:
+           curLength = length of cube (x direction)
+           curWidth = width of cube (y direction)
+----------------------------------------------------------------------*/
void makeCubePts(Dpoint3d *center) {
double halflength, halfwidth;

/* set points */
      halflength = curLength/2.0;
      halfwidth  = curWidth/2.0;
      shapePts[0].x = center->x-halflength;
      shapePts[0].y = center->y-halfwidth;
      shapePts[0].z = center->z;
      shapePts[1].x = center->x-halflength;
      shapePts[1].y = center->y+halfwidth;
      shapePts[1].z = center->z;
      shapePts[2].x = center->x+halflength;
      shapePts[2].y = center->y+halfwidth;
      shapePts[2].z = center->z;
      shapePts[3].x = center->x+halflength;
      shapePts[3].y = center->y-halfwidth;
      shapePts[3].z = center->z;
      shapePts[4].x = shapePts[0].x;
      shapePts[4].y = shapePts[0].y;
      shapePts[4].z = shapePts[0].z;
      numShapePts = 5;
}

/*----------------------------------------------------------------------+
+   placeShape: To place a shape in the drawing.
+   NOTE:
+       This routine assumes that the following variables are set:
+           shapePts = array of shape points
+           numShapePts = number of points for shape
+           curHeight = height of cube (z direction)
+           instName = name to append to element
+----------------------------------------------------------------------*/
int placeShape() {
MSElement element;
Dpoint3d basePt, topPt;
MSElementDescr *dscr1, *dscr2;

/* create shape and add to design file */
      if(mdlShape_create(&element, NULL, shapePts, numShapePts, 0) == SUCCESS) {
               if(mdlElmdscr_new(&dscr1, NULL, &element) != SUCCESS) {
                      mdlOutput_error("Error Creating Descriptor for Shape!");
                      return(FALSE);
               }

               /* 1 = cap (SOLID), 0 = nocap (SURFACE) */
               mdlParams_setActive(1, ACTIVEPARAM_CAPMODE);

               /* project shape up (projected by distance from basePt to topPt) */
```

```
                basePt.x = basePt.y = basePt.z = topPt.x = topPt.y = topPt.z = 0.0;
                topPt.z = curHeight;
                if(mdlSurface_project(&dscr2, dscr1, &basePt, &topPt, NULL) != SUCCESS)
{
                        mdlOutput_error("Error Projecting Shape Surface!");
                        return(FALSE);
                }

                /* add instance name */
                addXDataName(&dscr2->el, instName);

                /* display and add final cube */
                mdlElmdscr_display(dscr2, 0, NORMALDRAW);
                curFilePos = mdlElmdscr_add(dscr2);

                /* tag element */
                addTag(curFilePos);

                /* debugging */
                elmDscr_show(dscr2, "cube - ");

                /* free descriptors */
                mdlElmdscr_freeAll(dscr1);
                mdlElmdscr_freeAll(dscr2);
        }
        else {
                mdlOutput_error("Error Creating Shape!");
                return(FALSE);
        }
        return(TRUE);
}

/*-------------------------------------------------------------------------+
+-------------------------------------------------------------------------*/
int boxwed3(Dpoint3d *pt1, double l, double w, double s, double h, double a) {
Dpoint3d center;
        curLength = l;
        curWidth  = w;
        curHeight = h - s;
        center.x = pt1->x + (l / 2.0);
        center.y = pt1->y;
        center.z = pt1->z + s;
        makeCubePts(&center);
        placeShape();
        return(TRUE);
}

/*-------------------------------------------------------------------------+
+-------------------------------------------------------------------------*/
Public int main (void) {
RscFileHandle rscHandle;
char *setP;

        /* open resource file */
        mdlResource_openFile(&rscHandle, 0, FALSE);

        /* load the command table */
        if (mdlParse_loadCommandTable(NULL) == NULL)
                mdlOutput_printf(MSG_STATUS, "Error Loading Command Table");

        /* Publish the dialog item hooks */
```

```
    mdlDialog_hookPublish(sizeof(uHooks)/sizeof(DialogHookInfo), uHooks);

    /* open main dialog box */
    mdlDialog_open(NULL, DIALOGID_CADTALK);

    /* Initialize the C Expression environment and publish our variables */
    /* to the environment so the dialog box mananger can access them     */
    setP = mdlCExpression_initializeSet (VISIBILITY_DIALOG_BOX, 0, FALSE);

    mdlDialog_publishBasicVariable(setP, &floatType, "dialogx1", &dialogx1);
    mdlDialog_publishBasicVariable(setP, &floatType, "dialogx2", &dialogx2);
    mdlDialog_publishBasicVariable(setP, &floatType, "dialogx3", &dialogx3);
    mdlDialog_publishBasicVariable(setP, &floatType, "dialogx4", &dialogx4);
    mdlDialog_publishBasicVariable(setP, &floatType, "dialogy1", &dialogy1);
    mdlDialog_publishBasicVariable(setP, &floatType, "dialogy2", &dialogy2);
    mdlDialog_publishBasicVariable(setP, &floatType, "dialogy3", &dialogy3);
    mdlDialog_publishBasicVariable(setP, &floatType, "dialogy4", &dialogy4);
    mdlDialog_publishBasicVariable(setP, &floatType, "dialogz1", &dialogz1);
    mdlDialog_publishBasicVariable(setP, &floatType, "dialogz2", &dialogz2);
    mdlDialog_publishBasicVariable(setP, &floatType, "dialogz3", &dialogz3);
    mdlDialog_publishBasicVariable(setP, &floatType, "dialogz4", &dialogz4);
}

/*----------------------------------------------------------------------+
|       Command Handling routines                                       |
+----------------------------------------------------------------------*/
/*----------------------------------------------------------------------+
+----------------------------------------------------------------------*/
void getWord(char **args, char *word) {
char *temp;
    if (*args == NULL) return;
    strcpy(word, *args);
    temp = strchr(word, ' ');
    if (temp != NULL) *temp = '\0';
    /* printf("word %s\n", word); */
    *args = strchr(*args, ' ');
    if (*args == NULL) return;
    while (**args == ' ') ++*args;
}

/*----------------------------------------------------------------------+
+----------------------------------------------------------------------*/
void getFloatArg(char **args, double *x) {
    if (*args == NULL) return;
    *x = atof(*args);
    *args = strchr(*args, ' ');
    if (*args == NULL) return;
    while (**args == ' ') ++*args;
    /* printf("x %f, args %s\n", *x, *args); */
}

/*----------------------------------------------------------------------+
+----------------------------------------------------------------------*/
void getULongArg(char **args, ULong *x) {
    if (*args == NULL) return;
    *x = atol(*args);
    *args = strchr(*args, ' ');
    if (*args == NULL) return;
    while (**args == ' ') ++*args;
    /* printf("x %ld, args %s\n", *x, *args); */
}
```

```
void getLongArg(char **args, long *x) {
      if (*args == NULL) return;
      *x = atol(*args);
      *args = strchr(*args, ' ');
      if (*args == NULL) return;
      while (**args == ' ') ++*args;
      /* printf("x %ld, args %s\n", *x, *args); */
}

/*-------------------------------------------------------------------+
+-------------------------------------------------------------------*/
void getPointArg(char **args, Dpoint3d *pt) {
      getFloatArg(args, &pt->x);
      getFloatArg(args, &pt->y);
      getFloatArg(args, &pt->z);
      /* printf("pt %f %f %f\n", pt->x, pt->y, pt->z); */
}

/*-------------------------------------------------------------------+
| tagValue: retrieves the last tag_value set
+-------------------------------------------------------------------*/
Public cmdName void tagValue() cmdNumber CMD_GET_TAG {
char temp[10];
      mdlSystem_getenv(temp, "TAG_VALUE", 10);
      sprintf(resultBuf, "\"%s\"", temp);
      mdlSystem_putenv("CBRAIN_RESULTS", resultBuf);
}

/*-------------------------------------------------------------------+
| getElementID: given id (tag) will find element and                 |
|   set curElement and curFilePos                                    |
+-------------------------------------------------------------------*/
Public cmdName int getElementID(char *args) cmdNumber CMD_GET_ELEMENT_ID {
ULong tag;

      getULongArg(&args, &tag);
      if (mdlAssoc_getElement(&curElement, &curFilePos, tag, 0) != SUCCESS) {
               printf("element %d not found\n", tag);
               return 0; }
      if (curElement.ehdr.deleted) {
               printf("element %d found, but deleted\n", tag);
               return 0; }
      printf("element %d found at %ld, type=%d\n", tag, curFilePos,
      curElement.ehdr.type);
      curTag = tag;
      return 1;
}

/*-------------------------------------------------------------------+
| deleteGWObj: given id (tag) will find element and delete it.       |
+-------------------------------------------------------------------*/
Public cmdName deleteGWObj(char *args) cmdNumber CMD_DELETE_GW_OBJ {
      if (getElementID(args)) {
               mdlElement_undoableDelete(&curElement, curFilePos, TRUE);
               mdlSystem_putenv("CBRAIN_RESULTS", "T");
      }
}

/*-------------------------------------------------------------------+
| set_xdata: given id (tag) and instance name will find the element, |
|   set it's xdata, and return its instName                          |
```

```
+--------------------------------------------------------------------*/
Public cmdName set_xdata(char *args) cmdNumber CMD_SET_XDATA {
      if (getElementID(args)) {
               getULongArg(&args, &curTag);
               getWord(&args, instName);
               addXDataName(&curElement, instName);
               mdlElement_rewrite(&curElement, NULL, curFilePos);
               mdlSystem_putenv("CBRAIN_RESULTS", "T");
      }
}

/*--------------------------------------------------------------------+
| get_xdata: given id (tag) will find the element and return          |
|   its instName                                                      |
+--------------------------------------------------------------------*/
Public cmdName get_xdata(char *args) cmdNumber CMD_GET_XDATA {
      if (getElementID(args)) {
               getXDataName(&curElement);
               mdlSystem_putenv("CBRAIN_RESULTS", instName);
      }
}

/*--------------------------------------------------------------------+
|  alert: Display args in an okay/cancel box                          |
+--------------------------------------------------------------------*/
Public cmdName alert(char *args) cmdNumber CMD_ALERT {
      mdlDialog_openAlert(args);
}

/*--------------------------------------------------------------------+
+--------------------------------------------------------------------*/
Public cmdName void insert_text(char *args) cmdNumber CMD_INSERT_TEXT {
Dpoint3d origin;
char message[50];
MSElement element;

      if (args[0] == '\0') return;

      /* get origin */
      getPointArg(&args, &origin);

      /* get text to be placed on screen */
      getWord(&args, message);

      /* get name of instance */
      getWord(&args, instName);
      if (instName[0] == '\0') strcpy(instName, "NIL");

      mdlCurrTrans_begin();
      mdlCurrTrans_masterUnitsIdentity(TRUE);
      mdlText_create(&element, NULL, message, &origin, NULL, NULL, NULL, NULL);
      addXDataName(&element, instName);
      mdlElement_display(&element, NORMALDRAW);
      curFilePos = mdlElement_add(&element);
      addTag(curFilePos);
      mdlCurrTrans_clear();

      sprintf(resultBuf, "(\"%d\")", curTag);
      mdlSystem_putenv("CBRAIN_RESULTS", resultBuf);
}
```

```
/*--------------------------------------------------------------------+
+--------------------------------------------------------------------*/
Public cmdName void insert3dpolygon(char *args) cmdNumber CMD_INSERT_3D_POLYGON {
int i;
double temp;

      if (args[0] == '\0') return;

      /* get numShapePts */
      getFloatArg(&args, &temp);
      numShapePts = (int)temp;

      if (numShapePts > 8) return;

      /* get corner points */
      for (i=0; i < numShapePts; ++i)
               getPointArg(&args, &shapePts[i]);

      /* get curHeight */
      getFloatArg(&args, &curHeight);

      /* get instName */
      getWord(&args, instName);
      if (instName[0] == '\0') strcpy(instName, "NIL");

      /* set last point to first point to close shape and increase numShapePts */
      shapePts[numShapePts].x = shapePts[0].x;
      shapePts[numShapePts].y = shapePts[0].y;
      shapePts[numShapePts].z = shapePts[0].z;
      ++numShapePts;

      mdlCurrTrans_begin();
      mdlCurrTrans_masterUnitsIdentity(TRUE);
      /* mdlCurrTrans_translateOrigin(&shapePts[0]); */
      placeShape();
      mdlCurrTrans_clear();
}

/*--------------------------------------------------------------------+
+--------------------------------------------------------------------*/
Public cmdName void insert3dcube(char *args) cmdNumber CMD_INSERT_3D_CUBE {
Dpoint3d temp, origin, end;
double xLength, yLength, zLength, rotation;

      if (args[0] == '\0') return;

      /* get origin */
      getPointArg(&args, &origin);

      /* get xLength, yLength, zLength, and rotation */
      getFloatArg(&args, &xLength);
      getFloatArg(&args, &yLength);
      getFloatArg(&args, &zLength);
      getFloatArg(&args, &rotation);

      /* get instName */
      getWord(&args, instName);
      if (instName[0] == '\0') strcpy(instName, "NIL");

      mdlCurrTrans_begin();
      mdlCurrTrans_masterUnitsIdentity(TRUE);
```

```
    mdlCurrTrans_translateOrigin(&origin);
    mdlCurrTrans_rotateByAngles(0.0, 0.0, DEG_TO_RAD(rotation));
    temp.x = temp.y = temp.z = 0.0;
    boxwed3(&temp, xLength, yLength, 0, zLength, 0);
    mdlCurrTrans_clear();
}

/*----------------------------------------------------------------------+
+----------------------------------------------------------------------*/
Public cmdName void insert_line(char *args) cmdNumber CMD_INSERT_LINE {
MSElement element;

    if (args[0] == '\0') return;

    /* get end points of line */
    getPointArg(&args, &shapePts[0]);
    getPointArg(&args, &shapePts[1]);

    /* get instName */
    getWord(&args, instName);
    if (instName[0] == '\0') strcpy(instName, "NIL");

    /* if either shapePts[0].x or shapePts[1].x == -BIGINT, then open a dialog box
*/
    /* allowing the user specify the line.*/
    if (shapePts[0].x == -BIGINT || shapePts[1].x == -BIGINT) {
            dialogx1 = shapePts[0].x;
            dialogy1 = shapePts[0].y;
            dialogz1 = shapePts[0].z;
            dialogx2 = shapePts[1].x;
            dialogy2 = shapePts[1].y;
            dialogz2 = shapePts[1].z;
            mdlDialog_openModal(NULL, NULL, DIALOGID_LINEBOX);
            shapePts[0].x = dialogx1;
            shapePts[0].y = dialogy1;
            shapePts[0].z = dialogz1;
            shapePts[1].x = dialogx2;
            shapePts[1].y = dialogy2;
            shapePts[1].z = dialogz2;
    }

    mdlCurrTrans_begin();
    mdlCurrTrans_masterUnitsIdentity(TRUE);
    mdlLine_create(&element, NULL, shapePts);
    addXDataName(&element, instName);
    mdlElement_display(&element, NORMALDRAW);
    curFilePos = mdlElement_add(&element);
    addTag(curFilePos);
    mdlCurrTrans_clear();

    sprintf(resultBuf, "((\"%d\") (%4.2f %4.2f %4.2f) (%4.2f %4.2f %4.2f))",
            curTag,
            shapePts[0].x, shapePts[0].y, shapePts[0].z,
            shapePts[1].x, shapePts[1].y, shapePts[1].z);
    mdlSystem_putenv("CBRAIN_RESULTS", resultBuf);
}

/*----------------------------------------------------------------------+
+----------------------------------------------------------------------*/
Public cmdName void insert_rect_pts(char *args) cmdNumber CMD_INSERT_RECT_PTS {
MSElement element;
```

```
    if (args[0] == '\0') return;

    /* get vertices of rectangle */
    /* It is assumed that the 4 points form a rectangle */
    getPointArg(&args, &shapePts[0]);
    getPointArg(&args, &shapePts[1]);
    getPointArg(&args, &shapePts[2]);
    getPointArg(&args, &shapePts[3]);

    /* get instName */
    getWord(&args, instName);
    if (instName[0] == '\0') strcpy(instName, "NIL");

    /* if either shapePts[0].x or shapePts[1].x or shapePts[2].x or shapePts[3].x ==
-BIGINT, */
    /* then open a dialog box allowing the user specify the rectangle.*/
    if (shapePts[0].x == -BIGINT || shapePts[1].x == -BIGINT ||
        shapePts[2].x == -BIGINT || shapePts[3].x == -BIGINT) {
            dialogx1 = shapePts[0].x;
            dialogy1 = shapePts[0].y;
            dialogz1 = shapePts[0].z;
            dialogx2 = shapePts[1].x;
            dialogy2 = shapePts[1].y;
            dialogz2 = shapePts[1].z;
            dialogx3 = shapePts[2].x;
            dialogy3 = shapePts[2].y;
            dialogz3 = shapePts[2].z;
            dialogx4 = shapePts[3].x;
            dialogy4 = shapePts[3].y;
            dialogz4 = shapePts[3].z;
            mdlDialog_openModal(NULL, NULL, DIALOGID_RECTPTSBOX);
            shapePts[0].x = dialogx1;
            shapePts[0].y = dialogy1;
            shapePts[0].z = dialogz1;
            shapePts[1].x = dialogx2;
            shapePts[1].y = dialogy2;
            shapePts[1].z = dialogz2;
            shapePts[2].x = dialogx3;
            shapePts[2].y = dialogy3;
            shapePts[2].z = dialogz3;
            shapePts[3].x = dialogx4;
            shapePts[3].y = dialogy4;
            shapePts[3].z = dialogz4;
    }

    mdlCurrTrans_begin();
    mdlCurrTrans_masterUnitsIdentity(TRUE);
    mdlShape_create(&element, NULL, shapePts, 4, 0);
    addXDataName(&element, instName);
    mdlElement_display(&element, NORMALDRAW);
    curFilePos = mdlElement_add(&element);
    addTag(curFilePos);
    mdlCurrTrans_clear();

    sprintf(resultBuf, "((\"%d\") (%4.2f %4.2f %4.2f) (%4.2f %4.2f %4.2f)
                                (%4.2f %4.2f %4.2f) (%4.2f %4.2f %4.2f))",
            curTag,
            shapePts[0].x, shapePts[0].y, shapePts[0].z,
            shapePts[1].x, shapePts[1].y, shapePts[1].z,
            shapePts[2].x, shapePts[2].y, shapePts[2].z,
            shapePts[3].x, shapePts[3].y, shapePts[3].z);
```

```
    mdlSystem_putenv("CBRAIN_RESULTS", resultBuf);
}

/*----------------------------------------------------------------------+
+----------------------------------------------------------------------*/
Public cmdName void insert_rect_ht(char *args) cmdNumber CMD_INSERT_RECT_HT {
MSElement element;

    if (args[0] == '\0') return;

    /* get endpoints of base of rectangle */
    getPointArg(&args, &shapePts[0]);
    getPointArg(&args, &shapePts[1]);

    /* get height of rectangle */
    getFloatArg(&args, &curHeight);

    /* get instName */
    getWord(&args, instName);
    if (instName[0] == '\0') strcpy(instName, "NIL");

    /* if either shapePts[0].x or shapePts[1].x or height == -BIGINT,  */
    /* then open a dialog box allowing the user specify the rectangle. */
    if (shapePts[0].x == -BIGINT || shapePts[1].x == -BIGINT || curHeight == -BIGINT)
{
            dialogx1 = shapePts[0].x;
            dialogy1 = shapePts[0].y;
            dialogz1 = shapePts[0].z;
            dialogx2 = shapePts[1].x;
            dialogy2 = shapePts[1].y;
            dialogz2 = shapePts[1].z;
            dialogx3 = curHeight;
            mdlDialog_openModal(NULL, NULL, DIALOGID_RECTHTBOX);
            shapePts[0].x = dialogx1;
            shapePts[0].y = dialogy1;
            shapePts[0].z = dialogz1;
            shapePts[1].x = dialogx2;
            shapePts[1].y = dialogy2;
            shapePts[1].z = dialogz2;
            curHeight = dialogx3;
    }

    /* calculate two additional vertices */
    shapePts[2].x = shapePts[1].x;
    shapePts[2].y = shapePts[1].y;
    shapePts[2].z = shapePts[1].z + curHeight;
    shapePts[3].x = shapePts[0].x;
    shapePts[3].y = shapePts[0].y;
    shapePts[3].z = shapePts[0].z + curHeight;

    mdlCurrTrans_begin();
    mdlCurrTrans_masterUnitsIdentity(TRUE);
    mdlShape_create(&element, NULL, shapePts, 4, 0);
    addXDataName(&element, instName);
    mdlElement_display(&element, NORMALDRAW);
    curFilePos = mdlElement_add(&element);
    addTag(curFilePos);
    mdlCurrTrans_clear();

    sprintf(resultBuf, "((\"%d\") (%4.2f %4.2f %4.2f) (%4.2f %4.2f %4.2f)
                                 (%4.2f %4.2f %4.2f) (%4.2f %4.2f %4.2f))",
```

```
                curTag,
                shapePts[0].x, shapePts[0].y, shapePts[0].z,
                shapePts[1].x, shapePts[1].y, shapePts[1].z,
                shapePts[2].x, shapePts[2].y, shapePts[2].z,
                shapePts[3].x, shapePts[3].y, shapePts[3].z);
        mdlSystem_putenv("CBRAIN_RESULTS", resultBuf);
}

/*----------------------------------------------------------------------+
|                                                                       |
| name: insert_arc                                                      |
| arguments:                                                            |
|   point - the center of the circle that the arc is part of            |
|   point - starting point of the arc                                   |
|   angle - the sweep angle (radians)                                   |
|   string - name for the arc                                           |
| This assumes that arc lies flat in a plane parallel with z=0          |
+----------------------------------------------------------------------*/
Public cmdName void insert_arc(char *args) cmdNumber CMD_INSERT_ARC {
MSElement element;
Dpoint3d center, start;
double startang, sweepang, radius;

      if (args[0] == '\0') return;

      /* get center and two angles of arc */
      getPointArg(&args, &center);
      getPointArg(&args, &start);
      getFloatArg(&args, &sweepang);

      /* get instName */
      getWord(&args, instName);
      if (instName[0] == '\0') strcpy(instName, "NIL");

      /* if either center.x or start.x or sweepang == -BIGINT, then open a dialog box
*/
      /* allowing the user specify the arc.*/
      if (center.x == -BIGINT || start.x == -BIGINT || sweepang == -BIGINT) {
                dialogx1 = center.x;
                dialogy1 = center.y;
                dialogz1 = center.z;
                dialogx2 = start.x;
                dialogy2 = start.y;
                dialogz2 = start.z;
                dialogx3 = sweepang;
                mdlDialog_openModal(NULL, NULL, DIALOGID_ARCBOX);
                center.x = dialogx1;
                center.y = dialogy1;
                center.z = dialogz1;
                start.x = dialogx2;
                start.y = dialogy2;
                start.z = dialogz2;
                sweepang = dialogx3;
                }

      radius = sqrt((center.x - start.x)*(center.x - start.x) +
                      (center.y - start.y)*(center.y - start.y));
      startang = atan2(start.y - center.y, start.x - center.x);

      mdlCurrTrans_begin();
      mdlCurrTrans_masterUnitsIdentity(TRUE);
```

```
    mdlArc_create(&element, NULL, &center, radius, radius, NULL, startang, sweepang);
    addXDataName(&element, instName);
    mdlElement_display(&element, NORMALDRAW);
    curFilePos = mdlElement_add(&element);
    addTag(curFilePos);
    mdlCurrTrans_clear();

    /* return center point, start point, and sweep angle */
    sprintf(resultBuf, "((\"%d\") (%4.2f %4.2f %4.2f) (%4.2f %4.2f %4.2f) %f)",
            curTag,
            center.x, center.y, center.z, start.x, start.y, start.z, sweepang);
    mdlSystem_putenv("CBRAIN_RESULTS", resultBuf);
}
```

# Appendix B: Application Programming Interface (API)

The following list is the application programming interface (API) for the ACE/CAD interface. The API is composed of two separate sections of code. The first is code for ACE mainly written in LISP. This code provides the low-level CBrain functionality and handler functions for use by *BB-OBJECT*, the main object from which all others descend in ACE. These handlers are written in a generic form and are customized for use with specific CAD systems.

The second section of code is written for a specific CAD system. It contains functions that will be called by handlers written in ACE. These functions are mainly creation functions of the base CAD shapes and access functions for information stored within the CAD drawing. Some user interface code is also listed for creating menus and presenting information to the CAD user.

## ACE

Two low-level main functions are defined in CBrain for use in ACE:

*ac-eval* -

This function accepts an argument and sends it to the CAD system for evaluation. Typically this argument is a CAD command, such as "insert-line".

*ac-eval-text* -

This function is similar to ac-eval but accepts a string representation of the symbol. When this string is received in the CAD system, it is first converted to an symbol before evaluation.

Following are LISP functions and handlers defined for system startup and CAD interaction.

```
*****************************************************************************
BASE FUNCTIONS:
*****************************************************************************

-----------------------------------------------------------------------------
go-to-cad = Start the CAD system
Parameters:
     agent = Current agent working under
Note:
     The current agent name is passed to the CAD system.
     New instances created within CAD belong to this agent.
Usage:
     (go-to-cad 'user)

*****************************************************************************
GENERIC FUNCTIONS:
*****************************************************************************

-----------------------------------------------------------------------------
is-drawable-frame = Test if frame is CAD drawable
Parameters:
     frame = Frame name (either symbol or string)
Usage:
     (is-drawable-frame 'wall)
     (is-drawable-frame "wall")

-----------------------------------------------------------------------------
all-drawable = Return a list of all CAD drawable frames
Parameters:
     NONE
Usage:
     (all-drawable)

-----------------------------------------------------------------------------
cad-draw-all = Draw all instances of a frame (send cad-draw message to all
instances)
Parameters:
     frame = Frame name (either symbol or string)
Usage:
     (cad-draw-all 'wall)
     (cad-draw-all "wall")

*****************************************************************************
BB-OBJECT HANDLERS:
*****************************************************************************

-----------------------------------------------------------------------------
get-drawable = Get instance's drawable status
Parameters:
     NONE
Note:
     If value is 't than is drawable, else is not drawable.
Usage:
     (send-msg 'wall1 :get-drawable)

-----------------------------------------------------------------------------
```

```
set-drawable = Set instance's drawable status
Parameters:
     new status = Instance's new drawing status
Usage:
     (send-msg 'wall1 :set-drawable 't)

--------------------------------------------------------------------------
editable-slots = Return a list of slots the user may change
Parameters:
     NONE
Note:
     This handler is called from the CAD system to get a list of slots
     the user may change.
Usage:
     (send-msg 'wall1 :editable-slots)

**************************************************************************
BB-OBJECT REDIRECTION HANDLERS:
**************************************************************************

--------------------------------------------------------------------------
get-view-tag = Returns a list of tags that represent the CAD system
unique identifiers for the associated objects
Parameters:
     NONE
Usage:
     (send-msg 'wall1 :get-view-tag)

--------------------------------------------------------------------------
set-view-tag = Sets a list of tags that represent the CAD system
unique identifiers for the associated objects
Parameters:
     tags = List of unique identifiers
Usage:
     (send-msg 'wall1 :set-view-tag '(12))

--------------------------------------------------------------------------
clear-view-tag = Clears the list of tags that represent the CAD system
unique identifiers for the associated objects
Parameters:
     NONE
Usage:
     (send-msg 'wall1 :clear-view-tag)

--------------------------------------------------------------------------
cad-create = Initialize newly created object
Parameters:
     NONE
Note:
     This function is called when a new instance is created from within
     the CAD system.  It can be used to initialize slot values within
     the instance or any other purpose the user defines.
     When called, ACE will determine current CAD system, then call either
     the handlers:
             acad-create - For AutoCAD
             mstation-create - For Microstation
```

```
     The default implementation will do nothing.
Usage:
     (send-msg 'wall1 :cad-create)

------------------------------------------------------------------------
cad-draw = Draw object in CAD system
Parameters:
     NONE
Note:
     This function is called to draw the ACE object within the CAD system.
     When called, ACE will determine current CAD system, then call either
     the handlers:
             acad-draw - For AutoCAD
             mstation-draw - For Microstation
     The default implementation will draw the object as a CAD text object
Usage:
     (send-msg 'wall1 :cad-draw)
```

## CAD System

Two low-level main functions are defined in CBrain for use in the CAD system:

*cl-eval* -

This function accepts an argument and sends it to ACE for evaluation. Typically this argument is an ACE command, such as "create-instance."

*cl-eval-text* -

This function is similar to cl-eval but accepts a string representation of the symbol. When this string is received in ACE, it is first converted to a symbol before evaluation.

### *AutoCAD*

The following are functions defined in AutoLisp for the manipulation of extended entity data of an AutoCAD object and the creation of generic graphical representations of objects.

```
************************************************************************
EXTENDED ENTITY DATA FUNCTIONS:
************************************************************************

------------------------------------------------------------------------
add-xdata = Add extended entity data instance name to the AutoCAD object
Parameters:
     entity name = AutoCAD entity object entity information
     instance name = String of instance name
Usage:
     (add-xdata (entlast) "wall1")
```

```
--------------------------------------------------------------------------------
get-xdata = Get extended entity data instance name from the AutoCAD object
Parameters:
     entity name = AutoCAD entity object entity information
Usage:
     (get-xdata (entlast))

********************************************************************************
OBJECT CREATION FUNCTIONS:
********************************************************************************

--------------------------------------------------------------------------------
insert-text = Add an AutoCAD text object
Parameters:
     pt = Insertion point
     text = Text to be inserted
Usage:
     (insert-text '(5 5 0) "hi")

--------------------------------------------------------------------------------
insert-line = Add an AutoCAD line object
Parameters:
     pt1 = Start insertion point
     pt2 = End insertion point
Usage:
     (insert-line '(0 0 0) '(10 10 0))

--------------------------------------------------------------------------------
insert-2d-polygon = Add a 2d polygon object (an AutoCAD pline object)
Parameters:
     corners = List of points defining the polygon
Usage:
     (insert-2d-polygon '((13000 -1800 0) (18000 -5900 0) (13000 -8000 0)))

--------------------------------------------------------------------------------
insert-3d-polygon = Add a 3d polygon object (an AutoCAD pface object)
Parameters:
     corners = List of points defining the polygon
     height = Height of polygon
Usage:
     (insert-3d-polygon '((13000 -1800 0) (18000 -5900 0) (13000 -8000 0))
'500)

--------------------------------------------------------------------------------
insert-3dcube-main = Add a 3d cube object (an AutoCAD pface object)
Parameters:
     pt = Edge center point of cube
     length = Length of cube
     width = Width of cube
     height = Height of cube
     angle = Angle of cube
Note:
     Here is a detailed description of parameter relations:
pt is origin, length(x), width(y), height(z), a is rotation angle(z rotation
in AUNITS mode)
pt is smallest x, median y
```

```
ex. of draw:
   -------------------------------
   |                             |
   |                             |
   * < = pt                      |
   |                             |
   |                             |
   -------------------------------
Usage:
     (insert-3d-cube-main '(0 0 0) '10 '1 '8 '0)
```

### *Microstation*

Microstation uses the same functions for interaction with the CBrain communication interface. In addition, the following commands are defined in C programming language for the manipulation of extended entity data of a Microstation object and the creation of generic graphical representations of objects.

```
*****************************************************************************
EXTENDED ENTITY DATA COMMANDS:
*****************************************************************************

-----------------------------------------------------------------------------
get_tag = Retrieves the last tag (id) value set
Parameters:
     NONE
Usage:
     get_tag

-----------------------------------------------------------------------------
get_element_id = Display information of object matching id value
Parameters:
     id = id of object to display information on
Usage:
     get_element_id 12

-----------------------------------------------------------------------------
delete_gw_obj = Delete object matching id value
Parameters:
     id = id of object to delete
Usage:
     delete_gw_obj 12

-----------------------------------------------------------------------------
set_xdata = Set object's extended entity (user) data
Parameters:
     id = id of object to set extended entity (user) data
     instance name = String containing instance name to set
Usage:
     set_xdata 12 WALL1

-----------------------------------------------------------------------------
get_xdata = Get extended entity (user) data of object matching id value
```

```
Parameters:
     id = id of object to get extended entity (user) data of
Usage:
     get_xdata 12

------------------------------------------------------------------------------
alert = Display string in an okay/cancel dialog box
Parameters:
     text = String to display
Usage:
     alert THIS IS A TEST

******************************************************************************
OBJECT CREATION COMMANDS:
******************************************************************************

------------------------------------------------------------------------------
insert_text = Place a text object
Parameters:
     origin = Origin of text object
     text string = String for text object
     optional instance name = String containing instance name to set in
extended            entity (user) data
Usage:
     insert_text 0 0 0 WALL1 WALL1

------------------------------------------------------------------------------
insert_3d_polygon = Place a 3d polygon object
Parameters:
     Number points = Number of points in polygon object
     points = Listing of points in polygon object (multiple up to number given)
     height = Height for polygon object
     optional instance name = String containing instance name to set in
extended            entity (user) data
Usage:
     insert_3d_polygon 3 0 0 0 10 0 0 0 10 0 20 OBJ1

------------------------------------------------------------------------------
insert_3d_cube = Place a 3d cube object
Parameters:
     origin = Origin of cube object
     x length = X length of cube object
     y length = Y length of cube object
     z length = Z length of cube object
     rotation = Rotation of cube object
     optional instance name = String containing instance name to set in
extended            entity (user) data
Usage:
     insert_3d_cube 0 0 0 10 10 20 0 OBJ1

------------------------------------------------------------------------------
insert_line = Place a line object
Parameters:
     point1 = First point of line object
     point2 = Second point of line object
```

```
     optional instance name = String containing instance name to set in
extended            entity (user) data
Usage:
     insert_line 0 0 0 10 10 0 OBJ1

------------------------------------------------------------------------------
insert_rect_pts = Place a rectangular object
Parameters:
     point1 = First point of rectangular object
     point2 = Second point of rectangular object
     point3 = Third point of rectangular object
     point4 = Fourth point of rectangular object
     optional instance name = String containing instance name to set in
extended            entity (user) data
Usage:
     insert_rect_pts 0 0 0 10 10 0 0 0 10 10 10 10 OBJ1

------------------------------------------------------------------------------
insert_rect_ht = Place a rectangular object
Parameters:
     point1 = First point of rectangular object
     point2 = Second point of rectangular object
     height = Height for rectangular object
     optional instance name = String containing instance name to set in
extended            entity (user) data
Usage:
     insert_rect_ht 0 0 0 10 10 0 10 OBJ1

------------------------------------------------------------------------------
insert_arc = Place an arc object
Parameters:
     center = Center point of arc object
     starting point = Starting point of arc object
     sweep angle = Sweep angle of arc (in radians)
     optional instance name = String containing instance name to set in
extended            entity (user) data
Usage:
     insert_arc 0 0 0 10 0 0 0.5 OBJ1
```

# Acronyms and Abbreviations

| | |
|---|---|
| ACE | Agent Collaborative Environment |
| AEC | architecture/engineering community |
| API | application programming interface |
| CAD | computer-aided drafting |
| CRaDA | Cooperative Research and Development Agreement |
| DDE | Dynamic Data Exchange |
| DLL | Dynamic Link Library |
| LCS | local coordinate system |
| MDL | Microstation Development Language |
| OLE | Object Linking and Embedding |

## DISTRIBUTION

Chief of Engineers
ATTN: CEHEC-IM-LH (2)
ATTN: CEHEC-IM-LP (2)
ATTN: CECC-R
ATTN: CERD-L
ATTN: CEMP-ET (2)

Defense Tech Info Center 22304
ATTN: DTIC-O (2)

10
1/96